OSGi WireAdminService in der Praxis

Dipl.-Ing. (BA) Johannes Plaßmann
Siemens IT Solutions and Services GmbH

C-LAB Report

Vol. 10 (2011) No. 01

Cooperative Computing & Communication Laboratory

ISSN 1619-7879

C-LAB ist eine Kooperation
der Universität Paderborn und der Siemens IT Solutions and Services GmbH
www.c-lab.de
info@c-lab.de

C-LAB Report

Herausgegeben von
Published by

Dr. Wolfgang Kern, Siemens IT Solutions and Services GmbH
Prof. Dr. Franz-Josef Rammig, Universität Paderborn

Das C-LAB - Cooperative Computing & Communication Laboratory - leistet Forschungs- und Entwicklungsarbeiten und gewährleistet deren Transfer an den Markt. Es wurde 1985 von den Partnern Nixdorf Computer AG (nun Siemens IT Solutions and Services GmbH) und der Universität Paderborn im Einvernehmen mit dem Land Nordrhein-Westfalen gegründet.
Die Vision, die dem C-LAB zugrunde liegt, geht davon aus, dass die gewaltigen Herausforderungen beim Übergang in die kommende Informationsgesellschaft nur durch globale Kooperation und in tiefer Verzahnung von Theorie und Praxis gelöst werden können. Im C-LAB arbeiten deshalb Mitarbeiter von Hochschule und Industrie unter einem Dach in einer gemeinsamen Organisation an gemeinsamen Projekten mit internationalen Partnern eng zusammen.

C-LAB - the Cooperative Computing & Cooperation Laboratory - works in the area of research and development and safeguards its transfer into the market. It was founded in 1985 by Nixdorf Computer AG (now Siemens IT Solutions and Services GmbH) and the University of Paderborn under the auspices of the State of North-Rhine Westphalia.
C-LAB's vision is based on the fundamental premise that the gargantuan challenges thrown up by the transition to a future information society can only be met through global cooperation and deep interworking of theory and practice. This is why, under one roof, staff from the university and from industry cooperate closely on joint projects within a common research and development organization together with international partners. In doing so, C-LAB concentrates on those innovative subject areas in which cooperation is expected to bear particular fruit for the partners and their general well-being.

ISSN 1619-7879

C-LAB
Fürstenallee 11
33102 Paderborn
fon: +49 5251 60 60 60
fax: +49 5251 60 60 66
email: info@c-lab.de
Internet: www.c-lab.de

© Siemens IT Solutions and Services GmbH und Universität Paderborn 2010
Alle Rechte sind vorbehalten.
Insbesondere ist die Übernahme in maschinenlesbare Form sowie das Speichern in Informationssystemen, auch auszugsweise, nur mit schriftlicher Genehmigung der Siemens IT Solutions and Services GmbH und der Universität Paderborn gestattet.
All rights reserved.
In particular, the content of this document or extracts thereof are only permitted to be transferred into machine-readable form and stored in information systems when written consent has been obtained from Siemens IT Solutions and Services GmbH and the University of Paderbor

Inhaltsverzeichnis

1 Die OSGi Service Platform ... 4
1.1 Einführung ... 4
1.2 Service Component Registry: Das zentrale Element ... 5
1.3 WireAdminService ... 6
1.3.1 Einführung ... 6
1.3.2 Simple Producers und Consumers ... 7
1.3.3 Push und Pull / Demultiplexen ... 11
1.3.4 Beispiel für die Implementierung eines Consumers ... 14
1.3.5 Composite Producers und Consumers ... 15
1.3.6 Praktisches Beispiel ... 17
2 Praktische Anwendung im Projekt Robot2Business (R2B) ... 20
2.1 Einführung ... 20
2.2 Demonstrator ... 20
2.2.1 Hardware ... 20
2.2.2 Softwarestack Member-Instanz ... 20
2.3 WireAdminService im Einsatz ... 21
3 Fazit ... 25
4 Referenzen ... 27

1 Die OSGi Service Platform

1.1 Einführung

Die OSGi Service Platform definiert ein Softwaresystem, welches mittels eines Komponentenmodells das Erstellen von dynamischen Modulen (Bundles) und Diensten (Services) inklusive deren Versionierung ermöglicht. Diese Module werden in einer ServiceRegistry verwaltet. Die Bundles werden i.d.R. als Dienste ausgeprägt: Implementierungsdetails werden vor anderen Bundles verborgen, lediglich die Diensteschnittstelle wird veröffentlicht und in der ServiceRegistry hinterlegt.

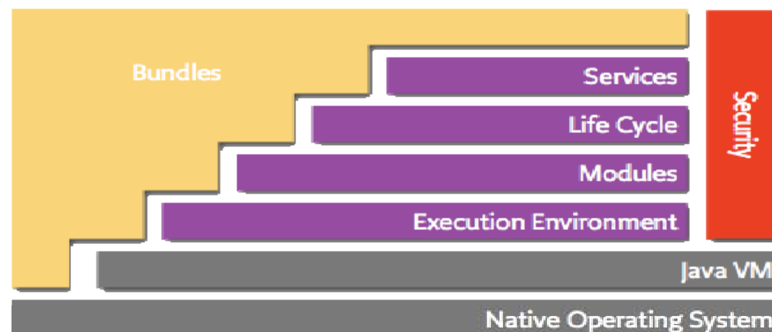
Diese Dienste können von anderen Bundles abgefragt werden und entsprechend zur Laufzeit mit angebunden werden („publish-find-bind"). Aufgrund der feingranularen Komponentisierung der Funktionalitäten mittles Bundles können je nach Zielsystem angepaßte und erweiterbare Anwendungen erstellt werden, die zudem noch speicheroptimiert sind. In einem ServiceContainer können die Bundles dynamisch gestartet, gestoppt, installiert und deinstalliert werden, ohne dass dazu der ServiceContainer in einen anderen Betriebszustand gebracht werden muss. Die Plattform basiert auf der Programmiersprache Java und macht es daher betriebssystemunabhängig. Die ServicePlatform ist bei verschiedenen OpenSource und kommerziellen Anbietern verfügbar.

Aufgrund der Modularisierung und der vom System mitgebrachten Dynamik mittels konsequenter Anwendung des Komponentenmodells wird die OSGi-Service-Platform nicht nur in eingebetteten Systemen, sondern zunehmend auch in Enterprise Systemen eingesetzt.

Neben der allgemeinen Plattform definiert die OSGi-Technologie auch Standarddienste, die ermöglichen sollen, dass Anwendungen auf einer standardisierten Infrastruktur aufsetzen können. Sie bringen so gesehen eine gewisse Grundfunktionalität mit. Zu den Standarddiensten gehörten z.B. LogService (Logging von Betriebsdaten), HttpService (WebServer Funktionalitäten), UserAdminService (Benutzer- und Gruppenverwaltung), EventAdminService (Anbieten von Eventing) und andere. Ein solcher Dienst ist auch der WireAdminService, der Verbindungen von Bundles über ein sogenanntes Wire ermöglicht und damit eine „Verdrahtung" von Services (WiringTopology) zur Laufzeit erstellt und betreibt. Über diese Wires können Bundles im Sinne von Produzenten (Producers) und Verbrauchern (Consumers) Informationen austauschen.

Diese Standarddienste sind, wie das gesamte OSGi-System, als Bundles verfügbar und können zur Laufzeit und nach Anforderungen entsprechend mit eingebunden werden.

Nachfolgendes Bild (aus [1]) zeigt das Layering von OSGi



- **Bundles:** Von Entwicklern geschriebene Komponenten (Die Summe aller Bundles machen eine Anwendung aus)
- **Services:** Dynamische Verbindung von Bundles mittel „publish-find-bind“
- **LifeCycle**: Funktionen zum Starten, Stoppen, Installieren und Deinstallieren von Bundles
- **Modules**: Beschreibt die Art und Weise, wie Bundles Funktionalität im- und exportieren können, bietet ein Versionierungssystem für Bundles an
- **Security**: Funktionen für Sicherheitsaspekte des Systems
- **ExecutionEnvironment**: Definiert, welche Klassen und Methoden in einer gegebenen Plattform verfügbar sind (wie z.B. Java2ME, Java2SE 5.0, Java2SE 6.0 oder andere Java Virtual Machines)

Technisch gesehen ist ein Bundle eine JAR-Datei, die neben Programmcode, Ressourcen wie Bilder, Bibliotheken sowie eine Manifest-Datei enthält. Diese beschreibt Deployment-Informationen des Bundles wie Import und Export von Klassen, Bundle-Name und –Version uvm. Da Bundles als Dienste agieren, ist OSGi die Realisierung von serviceorientierten Architekturen auf der Ebene von Softwarekomponenten.

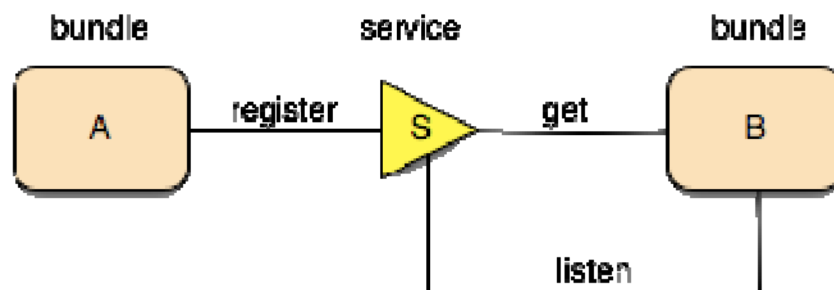
1.2 Service Component Registry: Das zentrale Element

“Don’t call us, we call you” Prinzip (Whiteboard Pattern)

Bundles bieten ihre Dienste über Schnittstellenbeschreibungen an. Diese Beschreibungen werden in einer zentralen Registry, der sog. Service Component Registry (SCR), hinterlegt, indem das entsprechende Bundle diese Information über einen BundleContext übergibt (Register). Zusätzlich zur Schnittstellenbeschreibung können auch noch weitere den Dienst beschreibende Elemente in Form von Key-Value-Paaren (Properties) mit hinterlegt werden. Indem ein Bundle einen Dienst registriert, zeigt es damit an, dass dieser nun für andere Bundles benutzbar ist.

Bundles, die einen entsprechenden Dienst benötigen, können ebenfalls über einen BundleContext bei der SCR anfragen, ob dieser eingetragen ist. Wenn ja, wird eine Referenz auf das entsprechende Bundle geliefert.

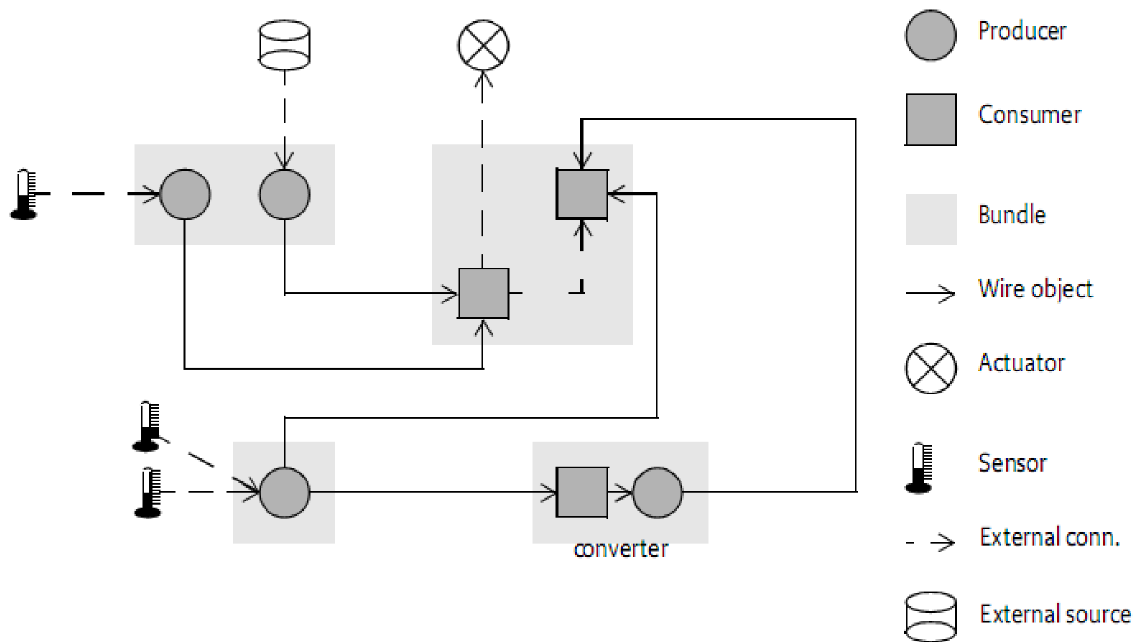
Ein Bundle kann sich auch bei der SCR als Listener registrieren, um benachrichtigt zu werden, wenn ein bestimmter Dienst verfügbar ist oder sich von der SCR abgemeldet hat. Dieses ergibt mehr Dynamik im Szenario einer erweiterbaren Anwendung. Durch Verwendung einer SCR können die Bundles entsprechend fein-granular in ihren Funktionalitäten entworfen werden. Eine Anwendung kann somit einfach und dynamisch um Funktionalitäten erweitert werden.



1.3 WireAdminService

1.3.1 Einführung

Bundles suchen Dienste über den Mechanismus wie oben beschrieben. Welche Dienste das sind und benötigt werden, ist in dem entsprechenden Bundle durch Programmcode hinterlegt. Der WireAdminService hingegen verbindet verschiedene Dienste z.B. über eine Konfigurationsdatei oder einen Anwendungsdialog (im Allgemeinen eine Managementapplikation). Das Wiring von Diensten erlaubt es, eine konfigurierbare Kooperation von Bundles aufzubauen, die dynamisch zur Laufzeit etabliert werden kann. Dabei wird das Konzept von Verbrauchern (Consumer) und Produzenten (Producer) angewandt, d.h. es gibt Dienste, die Daten erzeugen, und solche, die Daten entgegennehmen. Diese Dienste tauschen die Daten über sogenannte Wires aus. Typischerweise wird der WireAdminService dann eingesetzt, wenn ein System Sensoren und Aktoren enthält, die es zur Laufzeit zu erfassen und bedienen gilt.



Obiges Bild illustriert ein Szenario von Producern und Consumern (Entnommen aus [2])

Der WireAdminService implementiert die Schnittstelle **`org.osgi.service.wireadmin`** und läuft als Bundle im OSGi-ServiceFramework (z.B. von Framework Equinox : org.eclipse.equinox.wireadmin_1.0.0.v20080407.jar)

1.3.2 Simple Producers und Consumers

Um Bundles miteinander verdrahten zu können, müssen diese jeweils mittels einer Klasse die Schnittstellen **`org.osgi.service.wireadmin.Producer`** bzw. **`org.osgi.service.wireadmin.Consumer`** implementieren. Ein Producer erzeugt Daten und ein Consumer empfängt Daten.

Sobald ein Producer über eine Verbindung (Wire) mit einem Consumer verbunden wird (z.B. durch ein Managementbundle, das den WireAdminService kontaktiert), werden beide Beteiligte über die Callback-Funktion **`consumersConntected`** (für den Producer) bzw. **`producersConnected`** (für den Consumer) informiert, über welches Wire sie verbunden sind. D.h. jeder Partner muss sich explizit die jeweiligen Verbindungen merken. Ein Producer kann mittels des Wires den oder die Consumer über neue Werte informieren (**push**), ebenso kann ein Consumer ein oder mehrere mit ihm verbundene Producer um neue Wert anfragen (**poll**).

Ein Producer kann mit mehreren verschiedenen Consumern verbunden sein.

Producer und Consumer müssen sich bei der ServiceComponentRegistry mit den entsprechenden Properties anmelden, damit diese Dienste dann durch den WireAdminService gefunden und verknüpft werden können.

Folgende Properties müssen definiert werden:

- **Flavors**: Ein Array von Klassen, welche der entsprechende Producer über ein Wire schicken kann; für den Consumer, welche Klassen dieser handhaben kann („preferred data types")
- **Service.PID**: Eine systemweit eindeutige String-Kennung für den Producer oder Consumer; wird benötigt, damit der WireAdminService einen Producer mit einem Consumer verbinden kann: Die Service.PIDs werden apriori durch den Bundle-Entwickler vergeben.
- **ServiceDescription**: Beschreibungstext des entsprechenden Producers/Consumers

Eine Anmeldung eines Producers/Consumers mit den entsprechenden Properties erfolgt typischerweise in der **start**-Methode des entsprechenden Bundles:

Hier ein Beispiel für einen Producer:

```
public class Activator implements BundleActivator {

	public void start( BundleContext context ) {

		// build properties
		Properties props = new Properties();

		// the TestProducer sends Integers over the wire
		props.put( WireConstants.WIREADMIN_PRODUCER_FLAVORS,
				new Class[] { Integer.class } );
		props.put( Constants.SERVICE_PID, “TestProducer.PID“);
		props.put( Constants.SERVICE_DESCRIPTION, “The TestProducer”);

		// create a new TestProducer and register as Producer
		// tell SCR, that TestProducer offers interface
		// org.osgi.service.wireadmin.Producer
		context.registerService( Producer.class.getName(),
						new TestProducer(),
							props );

		// that’s all you’ve got to do to get the TestProducer running
	}
}
```

und hier für einen Consumer:

```
public class Activator implements BundleActivator {

	public void start( BundleContext context ) {

		// build properties
		Properties props = new Properties();

		// the TestConsumer understands Integers, Strings and
		// a TestClass (defined elsewhere)
		props.put( WireConstants.WIREADMIN_CONSUMER_FLAVORS,
				new Class[] { Integer.class,
					String.class,
						TestClass.class } );
		props.put( Constants.SERVICE_PID, “TestConsumer.PID“);
		props.put( Constants.SERVICE_DESCRIPTION, “The TestConsumer”);

		// create a new TestConsumer and register as Consumer
		// tell SCR, that TestConsumer offers interface
		// org.osgi.service.wireadmin.Consumer
		context.registerService( Consumer.class.getName(),
						new TestConsumer(),
							props );

		// that’s all you’ve got to do to get the TestConsumer running
	}
}
```

Wie ersichtlich, ist das Verfahren zur Registrierung für Producer und Consumer recht identisch. Producer und Consumer müssen lediglich einen Dienst bereitstellen und der SCR anzeigen, womit die Komplexität entsprechend klein gehalten werden kann, da keine zusätzlichen Funktionalitäten vonnöten sind. Ebenfalls werden Producer und Consumer entkoppelt, so dass sie den jeweiligen Partner nicht explizit kennen müssen. Dieses wird durch die Verwendung des Whiteboard-Patterns mittels der SCR erreicht.

Eine Management-Anwendung (z.B. implementiert in einem anderen Bundle) kann jetzt diese zu einem frei gewählten Zeitpunkt miteinander verbinden. Dieses geschieht, in dem sich diese Anwendung eine ServiceReference auf den WireAdminService erfragt und Producer und Consumer über die bekannten ServicePIDs verdrahtet. Die Methode **`wire()`** des WireAdminService sucht dann in der SCR nach den jeweiligen entsprechenden Diensten und ruft dann, sofern gefunden, die Methoden **`consumersConnected()`** des Producers und **`producersConnected()`** für den Consumer auf. Beiden Methoden wird das erzeugte Wire-Object als Parameter übergeben, damit sich Producer und Comsumer das neu erzeugte Wire merken können.

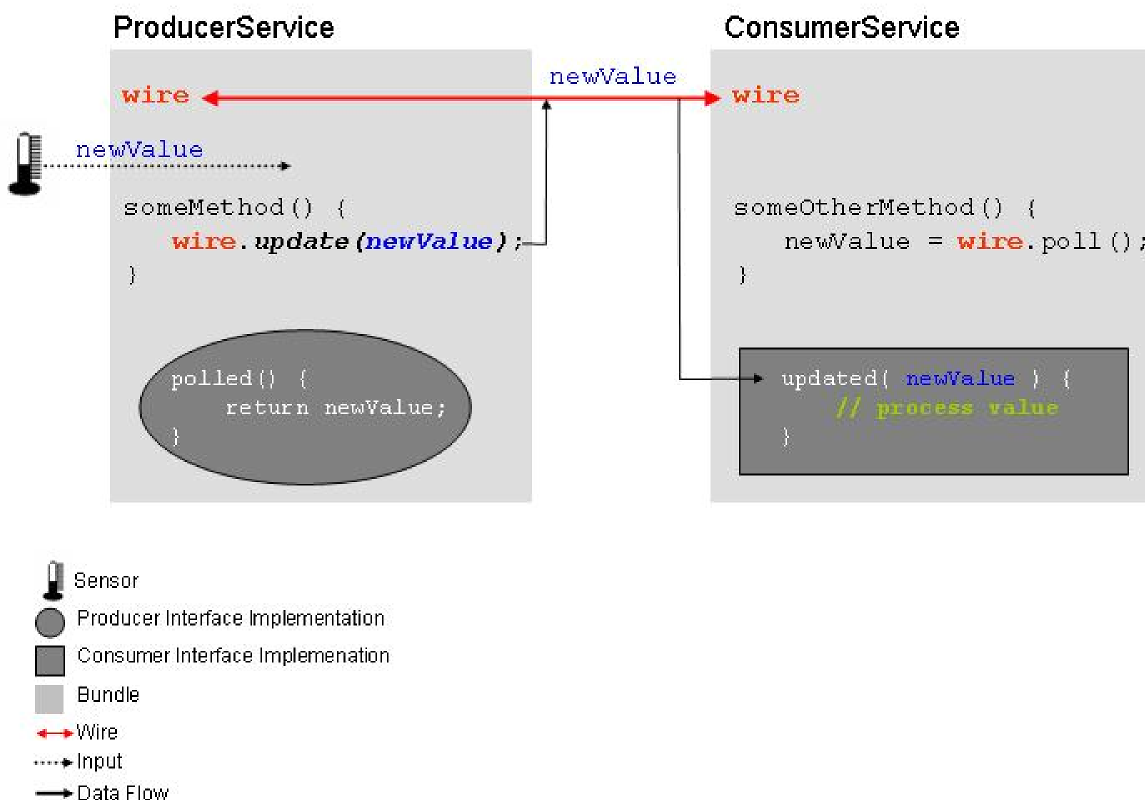
```
// get a service reference of the WireAdminService
ServiceReference sref = bundleContext.getServiceReference( WireAdmin.class.getName() );

WireAdmin wireAdmin = (WireAdmin) bundleContext.getService( sref );

// create a wire
// during the call the Producer and Consumer will be given an info
// that a new wire has been created via calls to connectedProducers and
// connectedConsumers
// this wire can be used elsewhere
Wire wire = wireAdmin.createWire( “TestProducer.PID”,
                                  “TestConsumer.PID,
                                  null );
// third parameter gives special properties to the wire, not needed here, so give a “null” value
```

1.3.3 Push und Pull / Demultiplexen

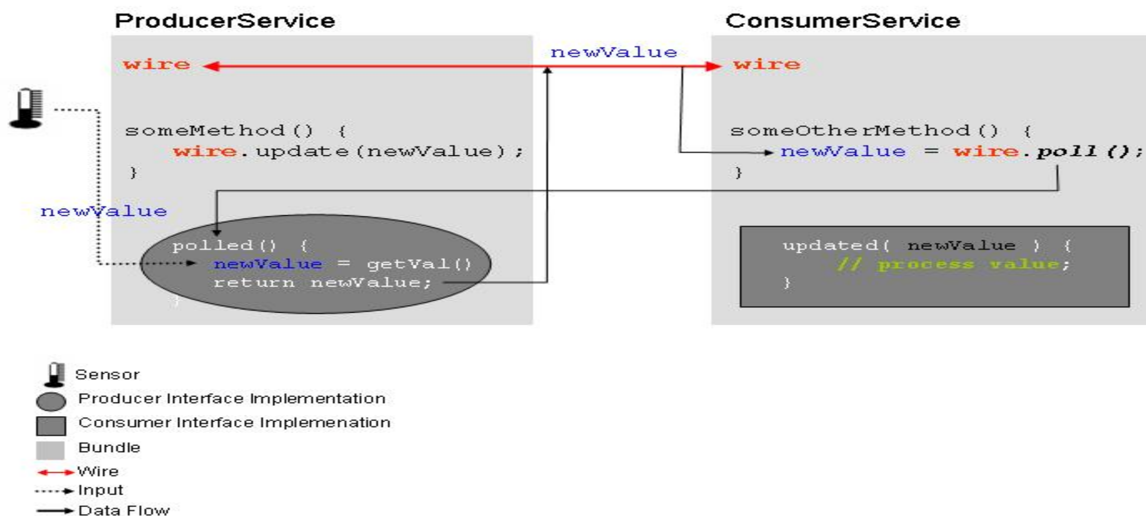
Sobald Producer und Consumer verbunden sind (d.h. jedem liegt mindestens ein Wire vor), können Daten ausgetauscht werden.

Dieses geschieht, in dem entsprechende Methoden auf dem Wire aufgerufen werden.

Wenn ein Producer einen neuen Wert vorliegen hat, so ruft er die **`update()`**-Methode mit dem entsprechenden neuen Wert auf. Dieses bewirkt, dass die Methode **`updated()`** im entsprechend verbundenen Consumer aufgerufen wird. Der Consumer kann dann diesen Wert prüfen und verarbeiten. (Push-Verfahren)



Der WireAdminService bietet auch ein symmetrisches Verfahren zum Push an: Wenn ein Consumer einen neuen Wert braucht, so ruft er die Methode **`poll()`** des Wire-Objektes auf. Dies bewirkt, dass beim Producer die Methode **`polled()`** aufgerufen wird, die dann einen aktuellen Wert zurückliefert. (Poll-Verfahren):



Ein Producer und ein Consumer werden über den WireAdminService mittels deren ServicePIDs verbunden. Über die Flavors zeigen die beteiligten Partner an, welche Art von Daten sie handhaben. Diese müssen nicht deckungsgleich sein, d.h., ein Producer kann Typen von Daten senden, die ein Consumer nicht handhaben kann. Auf diesen Umstand muss der Consumer Rücksicht nehmen und entsprechend reagieren können. Dieses geschieht, indem der Consumer explizit in der **`updated()`**-Methode nachfragt, ob es sich um ein Datum handelt, welches er unterstützt bzw. handhaben kann.

Beispiel für Demultiplexen im Consumer:

(Consumer unterstützt Flavors “Integer”, “String” und „TestClass“)

```
if ( data instanceof Integer ) {
        // process this integer value

} else if ( data instanceof String ) {
        // process this string value

} else if ( data instanceof TestClass ) {
        // process this TestClass value

} else {
        // no valid data found which is supported by this consumer
        // handle error
}
```

Beispiel für die Implementierung eines Producers:

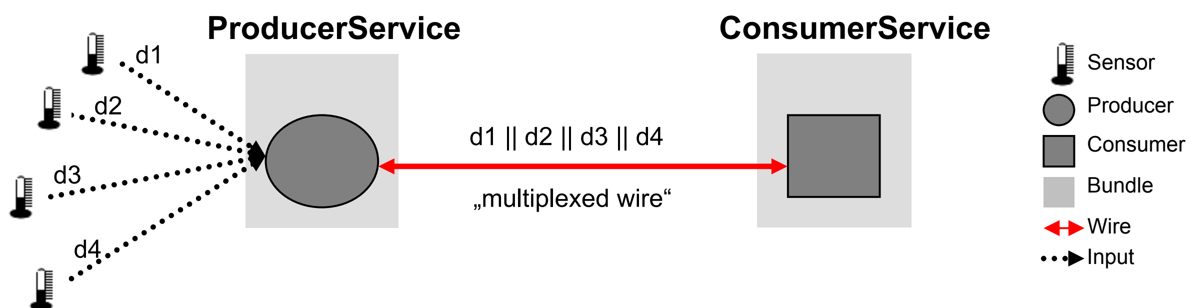
```
public class TestProducer implements org.osgi.service.wireadmin.Producer {

	// array for all connected wires
	Wires[] wires = null;

	// WireAdminService Producer implementation

	// callback for WireAdminService:
	// to inform about currently used wires
	// called each time a wire is created or deleted
	public void consumersConnected(Wire[] wires) {

		this.wires = wires;

		// add more code here…
	}

	// callback for WireAdminService:
	// called each time a consumer asks for new data
	// over the existing wire from producer (poll)
	public Object polled(Wire wire) {
		// return some data…
	}

	// method to push data to connected consumers over existing wires
	// is not part of the interface !!!
	// must be called each time new data needs to be pushed
	// as example: a new integer is being generated
	public void updateWires( Integer newData ) {

		for ( int i = 0; i < wires.length; i++ ) {

			// push data via update call of wire…
			wires[i].update( newData )
		}
	}

}
```

1.3.4 Beispiel für die Implementierung eines Consumers

```
public class TestConsumer implements org.osgi.service.wireadmin.Consumer {

        // array for all connected wires to this consumer
        Wires[] wires = null;

        // WireAdminService Consumer implementation

        // callback for WireAdminService:
        // to inform about currently connected wires of this consumer
        // called each time a new wire is created or an existing is deleted
        public void producersConnected(Wire[] wires) {

                // save all connected wires
                this.wires = wires;

                // add more code here…
        }

        // callback for WireAdminService:
        // called each time when there are changes of data
        // on connected wires (producer push)
        public void updated(Wire wire, Object data) {

                // check out, which type of data comes in
                // a kind of demultiplexing mechanism
                // check out for each flavor this consumer understands

                if (data instanceof Integer) {
                        // process this integer value
                } else if ( data instanceof String ) {
                        // process this string value
                } else if ( data instanceof TestClass ) {
                        // process this TestClass value
                } else {
                        // no valid data found which is supported by this consumer
                        // handle error
                }

        }


        // method to poll for new data over existing wires
        // is not part of the interface !!!
        // must be called when this consumer wants new data from connected producers
        public void pollWires() {

                for ( int i = 0; i < wires.length; i++ ) {

                        // poll data…
                        Object newData = wires[i].poll()

                        // handle data…
                }
        }
}
```

1.3.5 Composite Producers und Consumers

Normalerweise generiert ein Producer einen Typ von Information, ebenso wie ein Consumer einen Typ von Information verarbeitet. Diese werden als Dienste entsprechend in der SCR angemeldet. Bei vielen Producern sind entsprechend viele Dienste im SCR hinterlegt. Dies kann u.U. zu Performance-Verlusten führen. Ein anderes Szenario ist, dass ein Producer mehr als einen Typ von Informationen generiert, wie z.B. Außentemperatur als Ganzzahlwert, Füllstand Scheibenwischwasser als Ganzzahlwert, Status Fahrertür als Binärwert (offen/geschlossen) und Beifahrertür als Binärwert (offen/geschlossen). Ein Beispiel für einen solchen Producer ist ein Bundle, welches CANBus-Daten aufnimmt. CANBus-Daten liefern verschiedenste Status- und Wertinformationen, die über einen bestehenden Zugriffspunkt abgegriffen werden.



Um solche Daten unterscheidbar zu machen, führt der WireAdminService den Begriff des „Composite Producer“ bzw. „Composite Consumer“ ein.

Lösung für die Handhabung unterschiedlichster Daten ist die Verwendung einer sogenannten Envelope. Diese kapselt die unterschiedlich auftretenden Daten in einer vom WireAdminService definierten Datenstruktur. Sie dient dazu, dass jeweilige Datum für einen Consumer näher zu beschreiben. Da alle Daten mittels einer Envelope über das Wire übertragen werden, ist das einzige Flavor (Typen des Datums, welches über ein Wire gesendet werden) die Klasse Envelope (bzw. Ableitungen davon, da Envelope lediglich eine Schnittstelle definiert). Zusätzlich zum Flavor Envelope müssen Producer und Consumer sogenannte Scopes definieren. Ein Scope ist ein String, welches die Art eines Envelope-Objektes näher beschreibt, wie zum Beispiel „FüllstandDiesel“, „Außentemperatur“ oder „StatusTür“. Die Scopes werden vom Producer und Consumer vorab festgelegt und müssen ebenfalls durch die Properties mit registriert werden.

Beispiel für die Registrierung von Scopes und Flavors:

```
// build properties
String[] scopes = { "FuellstandDiesel", "Aussentemperatur", "StatusTuer" }
Class[] flavors = { Envelope.class }

Properties props = new Properties();

// define flavors
props.put( WireConstants.WIREADMIN_PRODUCER_FLAVORS, flavors );

// define scopes
props.put( WireConstants.WIREADMIN_PRODUCER_SCOPE, scopes );

// define servicePID
props.put( Constants.SERVICE_PID, "CANBusDataProvider.PID" );
props.put( Constants.SERVICE_DESCRIPTION, "The CANBusDataProvider" );

// register as composite producer
context.registerService( Producer.class.getName(),
                                             new CANBusDataProvider(),
                                                     props );
```

Werden nun Daten mittels einer Envelope ausgetauscht, so muss die entsprechende Envelope mit einem Scope gekennzeichnet werden. Dieser Scope muss in der Liste der bekannten Scopes beinhaltet sein, die der Producer anbietet.

Das Layout einer Envelope ist wie folgt (implementiert durch **`org.osgi.service.wireadmin.BasicEnvelope`**)

- **Value:**Wert des Datums; kann einen beliebigen Datentyp enthalten
- **Identification:** Dient zur Identifikation des Datums (Datenquelle); muss eindeutig sein; wird in gegenseitigem Einverständnis zwischen Producer und Consumer a priori vereinbart. I.d.R. ein String, kann aber auch beliebige Objekte beinhalten.
- **Scope:** Art/Kategorisierung des Datums; Beschreibung des Zwecks oder die Aufgabe dieses Datums; Datentyp String

1.3.6 Praktisches Beispiel

Ein Bundle erfasst die Daten „FüllstandDiesel“, „Außentemperatur“ und „StatusTür“ von einem CANBus. Das Bundle implementiert einen Producer (CANBusDataProvider), der die aufgenommenen Daten per Push versendet. Die Registrierung des Bundles mit den entsprechenden Properties erfolgt wie in dem obigen Code-Beispiel. Die Identifikation des Producers wird mit dem String „CANBusDataProvider“ gesetzt:

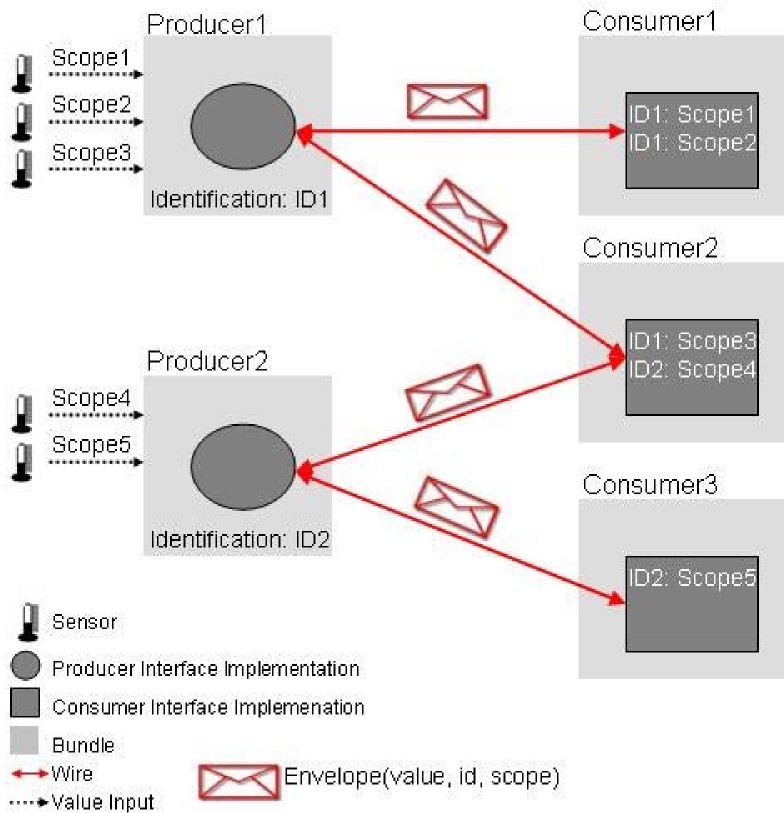
```
String[] scopes = { "FuellstandDiesel", "Aussentemperatur", "StatusTuer" };
String identification = "CANBusDataProvider";

for ( int i = 0; i < scopes.length; i++ ) {

        Object value = getCANBusData( scopes[ i ] );

        Envelope canBusData = new BasicEnvelope( value,
                                                            identification,
                                                                scopes[ i ] );

        wire.update( canBusData );
}
```

Ein Consumer kann dieses dann folgendermaßen verarbeiten:

```
public void updated( Wire wire, Object object ) {

        Envelope envelope = (Envelope) object;

        // handle upon producers
        if ( "CANBusDataProvider".equals( envelope.getIdentification() ) ) {

                // handle upon scopes of a given producer
                if ( "FuellstandDiesel".equals( envelope.getScope() ) ) {
                        // process Fuellstand information
                        ...
                } else if ( "Aussentemperatur".equals( envelope.getScope() ) ) {
                        // process Aussentemperatur
                        ...
                } else if ( "StatusTuer".equals( envelope.getScope() ) ) {
                        // process StatusTuer
                        ...
                }
                ...

        } else if ( "OtherProducer".equals( envelope.getIdentification() ) ) {
                ...
        }
}
```

Ein Beispiel, wie mehrere Producer und Consumer ein Zusammenspiel mittels der Identification und des Scopes arrangieren können, zeigt nachfolgende Grafik.

Zur Erklärung als Beispiel: Consumer1 verarbeitet die Identification ID1 (also von Producer1) und dann zwei von den möglichen drei Scopes dieses Producers)



2 Praktische Anwendung im Projekt Robot2Business (R2B)

2.1 Einführung

Das Projekt R2B ist ein vom Bundesministerium für Wirtschaft und Technologie gefördertes Forschungs- und Entwicklungsprojekt über den Zeitraum vom Oktober 2006 bis März 2010. Das Ziel ist die Einbindung und Vernetzung mobiler Maschinen in höherwertige Geschäftsprozesse und Dienstleistungsmodelle mittels Integration von teilautonomen Prozessen durch informationstechnische Verfahren.

Am Beispiel von (dynamischen) Prozessen aus der Landwirtschaft und dem Dienstleistungssektor soll eine Systembasis und ein Konzept geschaffen werden, welche flexibel und robust auf sich verändernde Abläufe, Umwelteinflüsse und Kommunikationsszenarien reagieren kann.

Für weitergehende Information sei auf [1] verwiesen.

2.2 Demonstrator

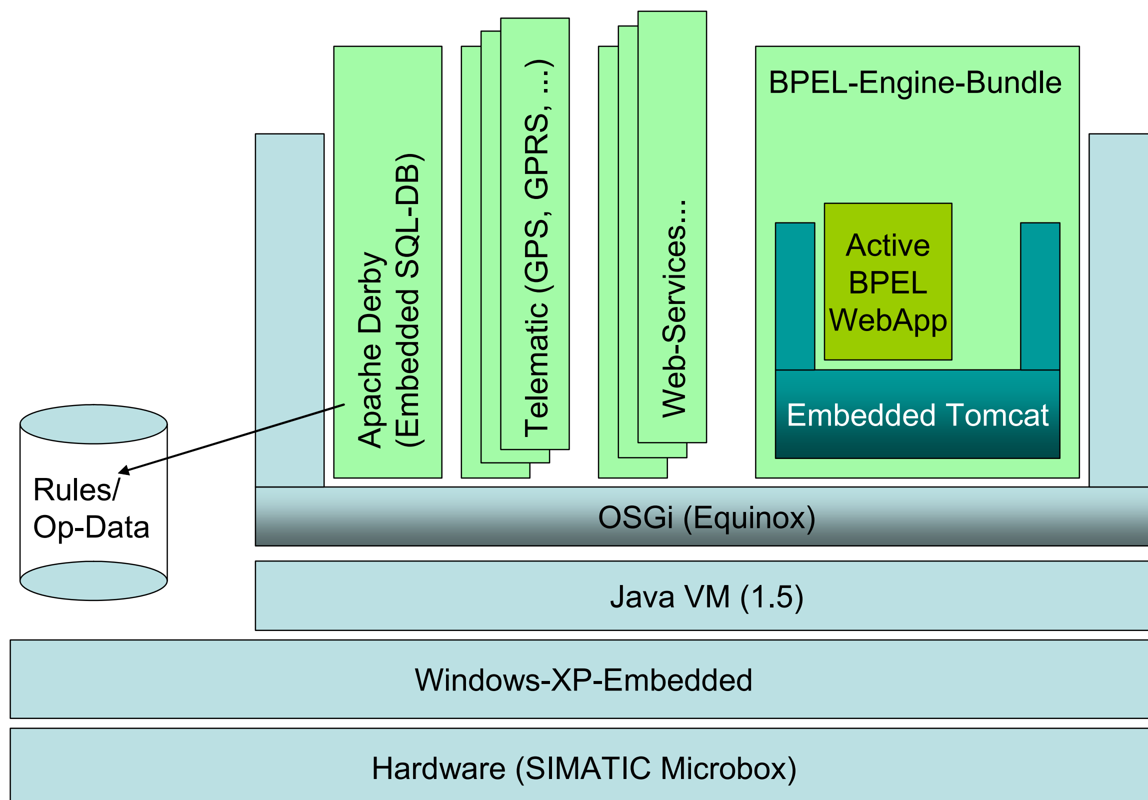
Im Rahmen des Projektes wurde ein Demonstrator für die Domäne Landwirtschaft entwickelt, der anhand ausgewählter Prozesse die prototypische Umsetzung der entstandenen Ideen und Konzepte zeigt.

2.2.1 Hardware

Die mobile Maschine (im R2B-Kontext als „Member-Instanz“ bezeichnet) wird durch einen Kindertraktor dargestellt, an dem entsprechende Hardware verbaut ist. Die Hardware besteht aus einem Embedded Industrie-PC (Simatic) mit 512MB Speicher sowie diversen Ein- und Ausgabeschnittstellen wie GPS, UMTS/GPRS, WLAN, iButtons und einem Touchscreen. Über diese verschiedenen Schnittstellen sind Kommunikation sowohl mit einem Backend-System als auch mit anderen mobilen Maschinen möglich. Nachfolgendes Bild zeigt den Demonstrator:

2.2.2 Softwarestack Member-Instanz

Die Applikationslogik wird durch eine Vielzahl von Bundles realisiert, die in einem OSGi-Container ablaufen. In diesem Container werden auch Bundles benutzt, die durch den WireAdminService vernetzt werden.



2.3 WireAdminService im Einsatz

Im Container werden Daten verschiedenster Art verarbeitet. Zum Beispiel Daten von einem CANBus, GPS-Daten, GPRS-Daten, iButton-Zustände und WLAN-Kommunikationsdaten.

Diese werden durch Provider-Bundles bereitgestellt. Die Daten aus diesen Bundles werden oftmals nicht nur von einem, sondern von mehreren Bundles gleichzeitig benötigt. Für solche Szenarien ist die Verwendung des WireAdminService von Vorteil, da eine Datenquelle (Producer) mehrere (interessierte) Datensenken (Consumer) mit Informationen versorgen kann.

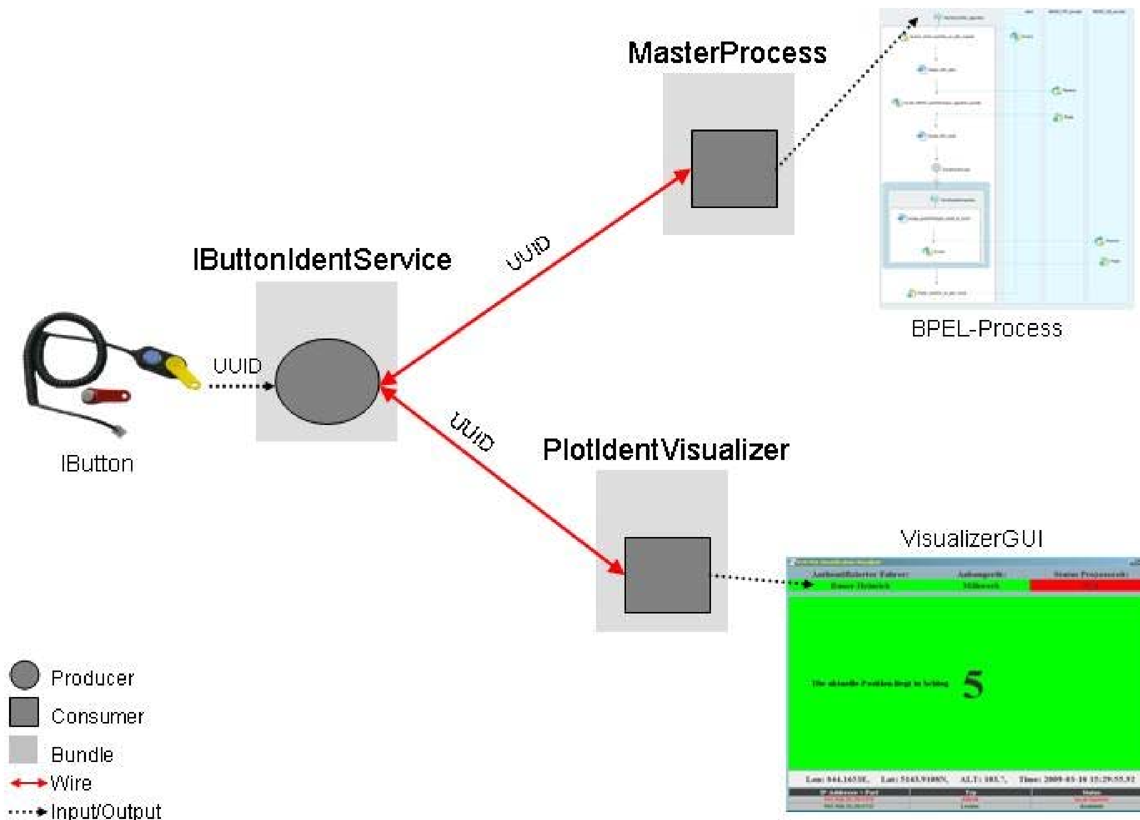
Die Steuerung der gesamten R2B-Applikation im OSGi-Container übernimmt ein MasterProcess-Bundle. Sobald gestartet, werden die entsprechend konfigurierten Wires erzeugt.

1. Szenario:

Die im Projekt verwendeten iButtons (siehe [4]) dienen der Identifikation des Fahrers und eines Anbaugerätes. Es sind spezielle elektronische Kontaktstecker, die mindestens eine UniqueID (UUID) enthalten und bei Kontakt mit der dafür vorgesehenen Aufnahme übertragen wird. Diese UUIDs können dann z.B. Fahrern und Geräten zugeordnet werden.

Wird ein Fahrer-iButton gesteckt, so wird diese Information (UUID) dem MasterProcess mitgeteilt (über das Wire zwischen IButtonIdentService und MasterProcess mittels Push-Verfahren) und dieser startet daraufhin einen BPEL-

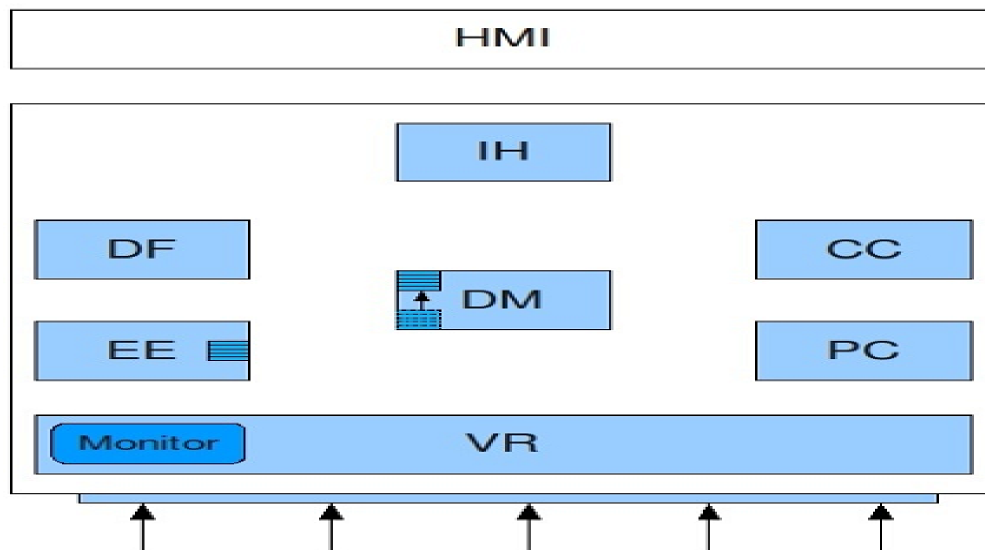
Prozess, welcher die durch den Prozess konfigurierten Betriebsdaten erfasst (z.B. Prozesszeit, Erntemenge, ...). Gleichzeitig wird die Fahrer-UUID ebenfalls zum PlotIdentVisualizer übertragen (über das Wire zwischen IButtonIdentService und PlotIdentVisualizer). Dieser wertet die UUID aus und stellt den entsprechenden Klartextnamen in der grafischen Oberfläche dar. Die VisualizerGUI ist eine Swing-Applikation im Bundle des PlotIdentVisualizers, welche Betriebsdaten und -zustände des Demonstrators in einer kleinen Oberfläche darstellt.



2. Szenario

Die im R2B-Memis-Container verwendete VisualizerGUI zur Anzeige von Betriebsdaten und –zuständen der mobilen Maschine soll auf Dauer durch eine andere, plattformunabhängige Lösung abgelöst werden. Dazu gibt es eine von der Firma CLAAS beauftragte Studie der FHDW Paderborn zum Thema Visualisierung von Zuständen landwirtschaftlicher Maschinen und beschreibt die Architektur und Abläufe dieses Human-Machine-Interfaces (HMI). Für weitere Information sei auf [5] verwiesen.

Die Architektur des HMI sieht wie folgt aus:



- IH: InteractionHandler
- DF: DisplayFactory
- EE: EscalationEngine
- CC: CommandCenter
- DM: DecisionMaker
- PC: PersistenceComponent
- VR: ValueReceiver

Die Umsetzung des HMI erfolgt ebenfalls innerhalb eines OSGi-Containers; die o.a. Komponenten werden dann jeweils als Bundle implementiert.

Die Komponente ValueReceiver hat die Aufgabe, die von externen Sensoren (repräsentiert durch die Pfeile im obigen Bild) gemeldeten Werte aufzunehmen und den anderen Komponenten zur Verfügung zu stellen.

Die Sensoren sollen die Werte möglichst selbständig liefern (Push-Verfahren).

In diesem Szenario eignet sich der Einsatz des WireAdminService besonders gut.

Jeder Sensor wird von einem Bundle repräsentiert, welches die Schnittstelle ISensorProducerService implementiert. Diese Schnittstelle garantiert, dass das Bundle zum einen als Producer auftritt, zum anderen steht dadurch auch die wichtige ServicePID (dient zum Erzeugen eines Wires) zur Verfügung. Jeder Sensor soll über ein Wire mit dem ValueReceiver verbunden werden und darüber die entsprechenden Informationen (z.B. Messwerte) liefern.

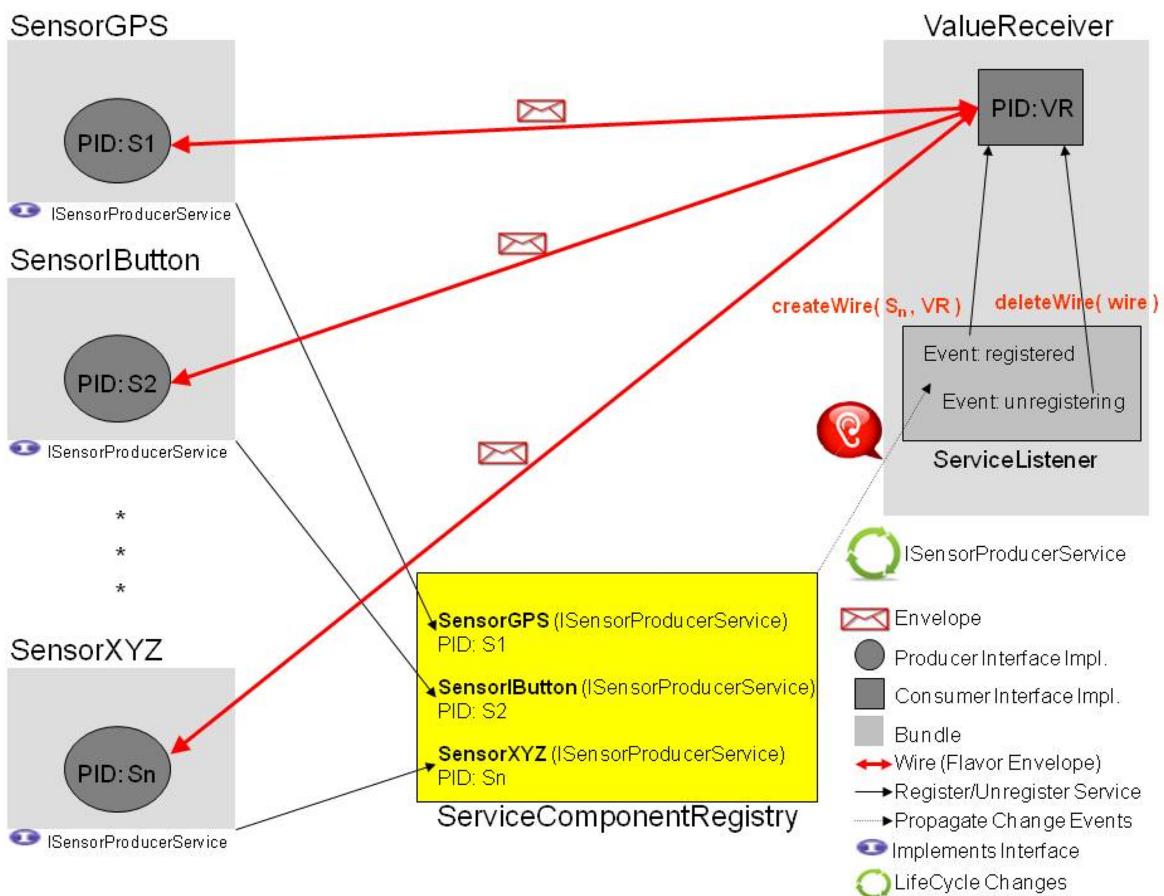
Jeder Sensor registriert sich mit der Schnittstelle „ISensorProducerService“ bei der ServiceComponentRegistry und zeigt damit an, dass sein Dienst zur Verfügung steht.

Der ValueReceiver seinerseits ist ebenfalls als Bundle ausgeführt und tritt als Consumer auf, der jedwelche Sensorwerte empfängt und verarbeitet oder wieter-reicht. Der ValueReceiver ist ebenfalls als ServiceListener aktiv: Er ist so konfiguriert, dass er mittelbar Benachrichtigungen von der ServiceComponentRegistry erhält, wenn Änderungen im LifeCycle von Diensten auftreten, die die Schnittstelle „ISensorProducerService“ implementieren.

Registriert sich ein Sensor in der SCR, so wird der Event **registered** ausgelöst und dem ServiceListener des ValueReceivers mitgeteilt. Der ValueReceiver kann dann den entsprechenden Sensor ermitteln und daraus die ServicePID ableiten. Mit dieser ServicePID des Sensors kann dann der ValueReceiver mittels des WireAdminService ein Wire über die Methode `createWire()` aufbauen.

Falls ein Sensor zur Laufzeit nicht mehr zur Verfügung steht, deregistriert er sich in der SCR. Daraufhin wird der Event **unregistering** ausgelöst und der ValueReceiver löscht dann mittels der Methode `deleteWire()` das Wire zwischen beiden Parteien.

Der ValueReceiver empfängt Werte sämtlicher Sensoren und tritt somit als Composite Consumer auf. Daher empfiehlt es sich, das Flavor eines jeden Wires als „Envelope“ auszuprägen.



3 Fazit

Der Einsatz des WireAdminServices bedarf vorheriger Überlegungen und des Verwendungszwecks.

Der WireAdminService eignet sich besonders für Szenarien, in denen Daten geliefert und an anderer Stelle verarbeitet werden müssen (Producer/Consumer-Muster). Hier sind besonders Aufgabenstellungen geeignet, die Sensoren beinhalten (Hardware-orientierte Systeme).

Für die Daten der Sensoren können dann Wrapper-Klassen bereitgestellt werden, die die Daten kapseln. Diese sensorspezifischen Daten können dann über vorgegebene Mechanismen im weiteren Verlauf durch Consumer-Komponenten verarbeitet werden.

Während normale OSGi-Dienste ihre Abhängigkeiten zur Entwicklungszeit festlegen, können Dienste, die den WireAdminService involvieren, zur Laufzeit konfiguriert werden. Dies geschieht üblicherweise mittels einer Konfigurationsdatei, die dann individuell pro Anwendungsfall gestaltet wird. Denkbar ist für die flexible Konfiguration z.B. auch ein UserInterface einer Managementapplikation, die erlaubt, die vorhandenen Elemente zu einem beliebigen Zeitpunkt miteinander zu verdrahten. Ein Ansatz diesbezüglich ist die Erweiterung der Apache Felix WebConsole um eine WireAdminService-Management- PlugIn (siehe [6])

Die Verwendung des WireAdminService ermöglicht eine n:m Datentopologie, d.h. es kann eine Anzahl von verschiedenen Producern mit einer anderen Anzahl von verschiedenen Consumern verbunden werden.

Zwar kann dieses auch durch Verwendung des normalen Service-Konzeptes von OSGi realisiert werden, ist aber mit einem ungleich höherem Entwicklungsaufwand verbunden, da die einzelnen Dienste dann sehr robust gestaltet werden müssen. Es erfolgt nicht automatisch eine Benachrichtigung, wenn ein daten-produzierender oder -konsumierender Dienst nicht mehr verfügbar ist. Bei Einsatz des WireAdminService-Konzept steht entweder eine Verbindung zur Verfügung oder nicht, ergo werden entsprechend Daten kommuniziert oder nicht.

Für die Entwicklung von Producern ist, wie im Verlaufe des Reports gezeigt, wenig Aufwand erforderlich. Die Erhebung von Daten kann man mittels der aufgezeigten Methoden fein-granular aufteilen (z.B. externen Geräten zuordnen, in dem für jedes Gerät ein eigenes Bundle entwickelt wird). Sorgfalt muss man beim Einsatz von Consumern walten lassen: Zwar ist die grundlegende Implementierung ähnlich einfach wie bei einem Producer, der Consumer übernimmt aber die Aufgabe des Demultiplexens der von einem oder mehreren Producern gesendeten Daten, so dass dort auf die richtige Zuordnung geachtet werden muss.

Zwar werden Producer und Consumer codetechnisch entkoppelt, d.h. ein expliziter Producer muss nicht einen dazugehörigen expliziten Consumer kennen; der Datenaustausch erfolgt aufgrund von kontraktierten Datenstrukturen und systemweit eindeutig festgelegten ServicePIDs. Funktional muss ein Consumer aber (vorab) vorbereitet sein, von welchen möglichen Producern Daten zu erwarten sind.

Daher verlangt der Einsatz des WireAdminService eine vorherige grundlegende Analyse der Struktur und Verknüpfung der zu erwartenden Daten.

Beim Einsatz von CompositeProducern bzw. CompositeConsumern ist die Verwendung einer Ableitung der vorgegebenen Datenstruktur Envelope notwendig. Hier können spezialisierte, auf den Anwendungsfall zugeschnittene Envelopes modelliert werden, die das Format der zu übertragenden Daten am besten unterstützen (z.B. durch Komfort-Methoden).

4 Referenzen

[1] OSGi Alliance:
http://www.osgi.org/About/WhatIsOSGi

[2] OSGi Alliance:
OSGi Service Platform Compendium, Release 4, Version 4.1, April 2007
Kapitel 108: Wire Admin Service Specification

[3] Robots2Business:
http://www.r2b-online.de

[4] iButton Product Page:
http://www.maxim-ic.com/products/ibutton/

[5] Thomas Kersting, FHDW Paderborn:
Entwicklung eines Human Machine Interfaces (HMI) zur Visualisierung des Status
Landwirtschaftlicher Maschinen, FHDW, Paderborn, Version 1.2, Stand 17.8.2009

[6] Apache Felix WebConsole:
http://felix.apache.org/site/apache-felix-web-console.html