



# MORE Middleware & Services

## Developers Guide

---

<b>Project Acronym:</b>	MORE
<b>Project Full Title:</b>	Network-centric Middleware for Group Communications and Resource Sharing across heterogeneous Embedded Systems
<b>Project Websites:</b>	<a href="http://www.ist-more.org">www.ist-more.org</a> <a href="http://services.ist-more.org">services.ist-more.org</a>
<b>Version:</b>	1.0
<b>Editors:</b>	Damien Lavaux (Thales), Jens Schmutzler, Constantin Timm (UniDo), Stefan Michaelis (PRO DV)
<b>Contributors:</b>	Alejandro Alonso (UPM), Gemma Power, Chris Foley, Chen Chen, Niall Donnely (WIT), Győző Karsai, Ákos Lévy (ALL), Damien Lavaux (Thales), Jens Schmutzler, Constantin Timm, Hanno Georg, Christian Lewandowski (UniDo), Stefan Michaelis, Peter Daum (PRO DV)



## Project Partners

<b>Nr.</b>	<b>Full Name</b>	<b>Short Name</b>	<b>Country</b>
1	PRO DV	PRO DV	Germany
2	Thales Communications S.A.	Thales	France
3	University of Dortmund, Communication Networks Institute & Embedded Systems Group	UniDo	Germany
4	Applied Logic Laboratory	ALL	Hungary
5	Waterford Institute of Technology, Telecommunications Software & System Group	WIT	Ireland
6	Technical University of Dresden	TUD	Germany
7	University of Debrecen	UniDe	Hungary
8	Universidad Politécnica de Madrid	UPM	Spain

# 1. INTRODUCTION

The Users guide introduced all the middleware functionalities currently available, and detailed for each of the middleware components how to use the functionalities they provide, and present a deeper view in the software architecture.

To complement the Users-guide, this Developer guide is intended to service developers or application developers that wants to extend some existing middleware component. Developers are suggested to look to the source code available for each middleware component on the set-up MORE SVN if they want to extend the framework.

## Table of Contents

1. INTRODUCTION .....	3
2. TUTORIAL – HOW TO DEVELOP A MORE SERVICE .....	6
3. MORE TOOLING SUPPORT .....	11
3.1.1. MORE WSDL User Documentation Generator .....	11
4. MIDDLEWARE COMPONENTS DEVELOPERS GUIDE .....	15
4.1. Core Management.....	15
4.1.1. Component design details.....	15
4.1.2. Future Developments .....	16
4.1.3. Generated Javadoc.....	16
4.2. Group Management .....	16
4.2.1. Component architecture/design details.....	16
4.2.2. Motivation for this particular design .....	17
4.2.3. Future Developments .....	17
4.2.4. Generated Javadoc.....	18
4.3. Resource Management .....	18
4.3.1. Component architecture/design details.....	18
4.3.2. Motivation for this particular design (Optional).....	19
4.3.3. Future Developments .....	19
4.3.4. Generated Javadoc.....	20
4.4. µSOA Proxy Service .....	22
4.4.1. Component architecture/design details.....	22
4.4.2. Future Developments .....	23
4.4.3. Generated Javadoc.....	23
4.5. Service Chaining.....	24
4.5.1. Component architecture/design details.....	24
4.5.2. Future Developments .....	26
4.5.3. Generated Javadoc.....	26
4.6. Communication Priorization Service.....	26
4.6.1. Processing of Messages.....	26
4.6.2. Sequence of Operations .....	27
4.6.3. Future Developments .....	27
5. MORE SERVICES DEVELOPERS GUIDE.....	29
5.1. Measurement Services.....	29
5.1.1. Component architecture/design details.....	29
5.1.2. Future Developments .....	29
5.2. Glucose Send Data Service .....	31
5.2.1. Component architecture/design details.....	31
5.2.2. Motivation for this particular design .....	33
5.2.3. Future Developments .....	33
5.3. Local Storage Service .....	33



5.3.1.	Component architecture/design details .....	33
5.3.2.	Future Developments .....	34
5.4.	Compression Service .....	34
5.4.1.	Component architecture/design details .....	34
5.4.2.	Future Developments .....	34
5.5.	Message Receiver Service .....	35
5.5.1.	Component architecture/design details .....	35
5.5.2.	Motivation for this particular design .....	35
5.5.3.	Future Developments .....	35

## 2. TUTORIAL – HOW TO DEVELOP A MORE SERVICE

This tutorial details the development process of a standard MORE service. The MORE service implementation procedure consists of the following steps:

- MORE Service Project Creation
- MORE Service Proxy and Interface Generation
- MORE Service Implementation
- MORE Service Export

The following text should help to create java applications using the MORE intermediate prototype.

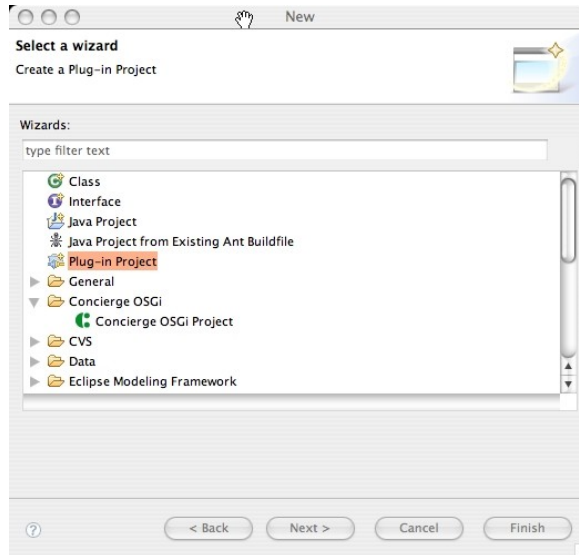
### **1. MORE Service Project Creation**

The Eclipse development platform comprises an extensible framework, tools and runtimes for building, deploying and managing software. The Eclipse Runtime is based on the OSGi specification. The OSGi implementation in Eclipse is named Equinox and is fully compliant to the OSGi R4.0 framework specification. Developing OSGi bundles can be performed using different tools. Here we'll focus on how to create a simple OSGi component compliant to the MORE service specification using Eclipse and the Eclipse PDE (Plugin Development Environment). In order to implement a new MORE service we'll need to have the following software installed in a new workspace:

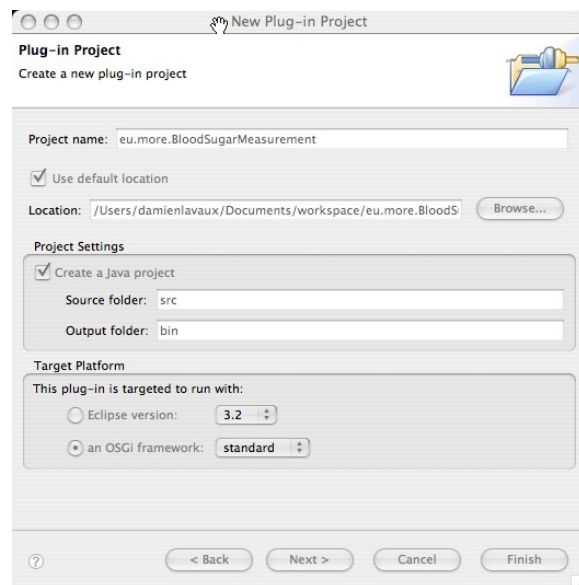
- ✓ Eclipse SDK 3.3
- ✓ The Eclipse Web Tools Platform (WTP)

The creation of a new MORE Service project comprises the following 4 steps:

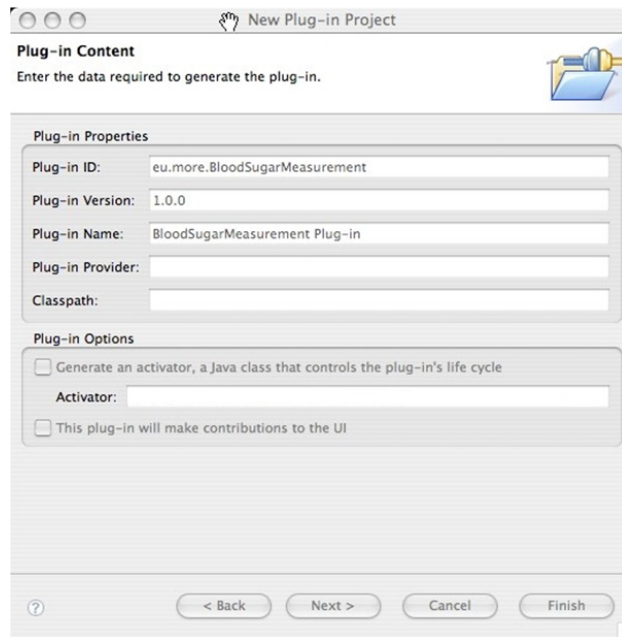
**Step 1: Create new plug-in project**



**Step 2: Insert project parameter**



Step 3: Generate a manifest



Step 4: Alter the manifest

The manifest must be changed to resolve the dependencies of the MORE service.

- ✓ **Import-Package:**
  - **org.osgi.framework**  
(Specify packages on which this plug-in depends without explicitly identifying their originating plug-in). This way our newly created bundle is independent of the OSGI service Platform implementation.
  - **org.osgi.service.packageadmin** and **org.osgi.util.tracker** These packages are needed to communicate with the MORE core instance.
- ✓ **Require-Bundle: eu.more.core**  
This is the only necessary bundle

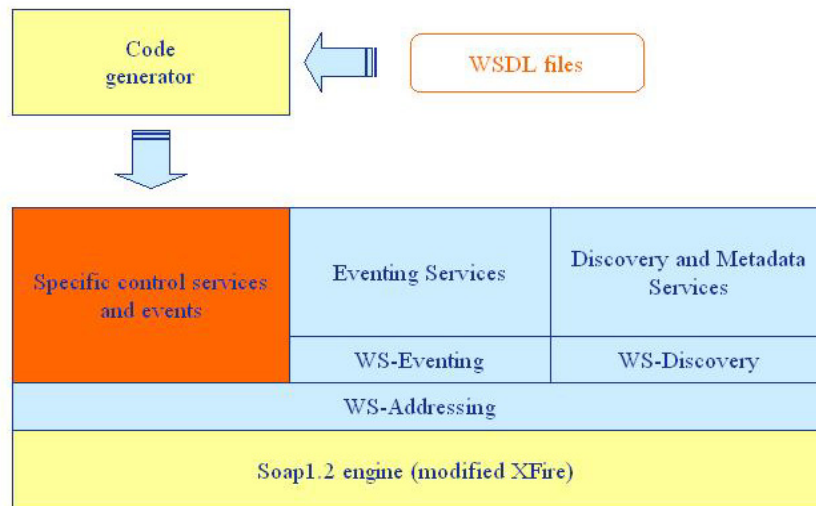
```

META-INF/MANIFEST.MF

Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: BloodSugarMeasurement Plug-in
Bundle-SymbolicName: eu.more.BloodSugarMeasurement
Bundle-Version: 0.1.0
Bundle-Activator: eu.more.bloodsugardevice.BloodSugarActivator
Bundle-Localization: plugin
Import-Package: org.osgi.framework;version="1.3.0", org.osgi.service.packageadmin;version="1.2.0", org.osgi.util.tracker;version="1.3.0",
Require-Bundle: eu.more.core
Bundle-Vendor: TAI
Bundle-ClassPath: ., lib/soda-JaxbBindingProvider.jar
    
```

## 2. MORE Service Proxy and Interface Generation

The DPWS4J toolkit can be used to generate proxies and interfaces from a WSDL file. For details on DPWS4J programming, the reader is referred to the DPWS4J toolkit User Guide (available on [www.soa4d.org](http://www.soa4d.org)). DPWS4J toolkit provides a code generator which generates Generic DPWS components from a WSDL file.



**Figure 1: DPWS code generation process**

### Basic characteristics of this toolkit are:

- Compliant with the DPWS specification
- Does not support the Discovery Proxy mechanism (optional)
- Based on the XFire open source SOAP engine
- Available on various Java toolkits:
  - o JDK 1.4
  - o JDK 1.5
  - o J2ME

Basic development principles are barely identical for DPWS services development and MORE intermediate prototype Services development. It can be shortly summarize to:

- Service interface design uses a WSDL file.
- Code generation :
  - o Generates set of java files.
  - o Eventually a jaxa/xml binding provider.
- Developing a client :
  - o Device / Service lookup
  - o Reading Device / Service metadata
  - o Service invocation
  - o Creating and managing subscriptions
  - o Receiving / Handling subscription events
  - o Handling Hello & Bye messages
- Implementing the services

- initializing service and device metadata

### 3. *MORE Service Implementation*

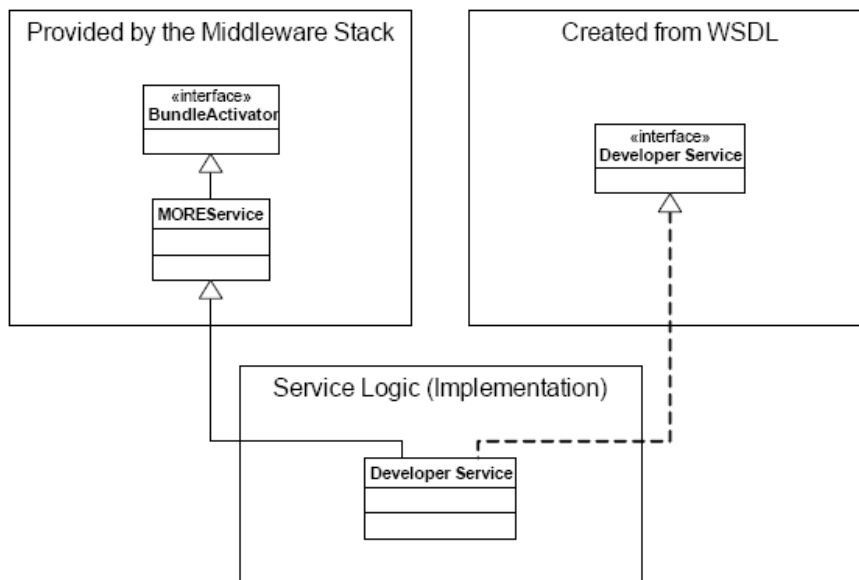
The CMS is responsible for inspecting the MORE Service Deploy Directory and for launching the present MORE Services. The MORE Core Management Service provides a CoreServer which registers all MORE services. Every MORE service must implement the service logic interface and must inherit from the MOREService class. The MOREService class is provided by the MORE consortium to enable a service developer to concentrate on the service logic implementation. The MORE middleware provides the MOREService class as the basic middleware class that manages all internal functions a service developer should not care about. The following functions are automatically derived from the MOREService:

- Publishing the service to the CORE
- Accessing local services on a device. This includes locating a service and invoking operations of the service.
- Providing of logging functionalities

```
public class BloodSugarService extends MOREService implements Measurement {
```

**Figure 2 : Example MORE Service Declaration**

An example MORE Service declaration and the inheritance structure are shown in Figure 2 and Figure 3.



**Figure 3: MORE inheritance**

#### 4. MORE Service Export

Exporting the created MORE services as OSGI bundle is the last step of the MORE service implementation process. The Eclipse PDE tools provide a convenient export wizard to perform this task. Moreover, it is possible to configure the target platform in eclipse for the bundles to be deployable on many different environments.

Use Eclipse Preferences .. > Plug-in Development > Target Platform and select the target project in your workspace as location.

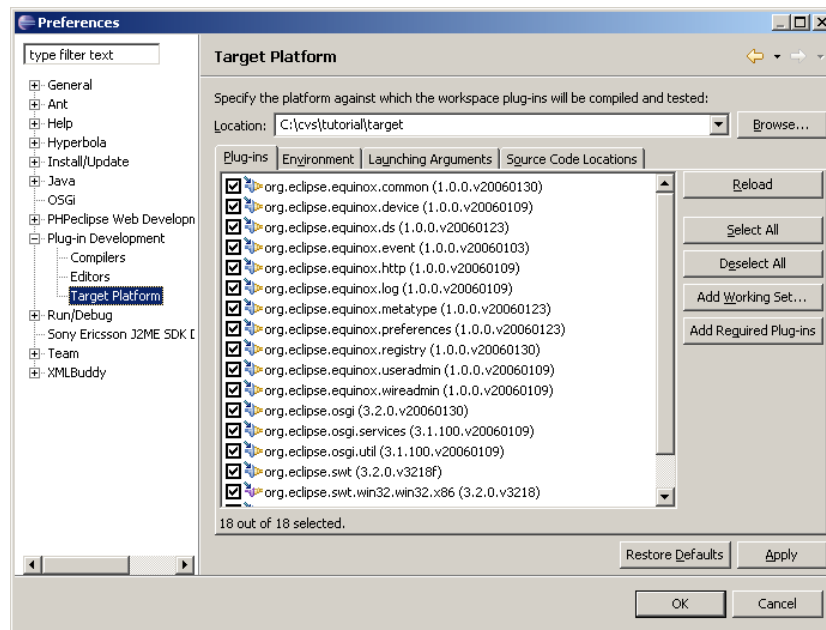


Figure 4: Screenshot of Eclipse Platform Setup

Thanks to the PDE Manifest editor export wizard, we now obtain a fully deployable bundle compacted as a .jar file representing a MORE intermediate prototype Service. (see directory /bundles of the delivered package for samples).

## 3. MORE TOOLING SUPPORT

### 3.1.1. MORE WSDL User Documentation Generator

A major important document for MORE middleware users, operators and of course for developers as well is access to the operational functionality of each service. Needed information is provided in machine readable form in the Web Service Description Language files provided for each service. For the human user these files are typically harder to read and

understand, although several integrations into development environments like Eclipse and Netbeans exist. Nevertheless, for the first understanding of the access methodologies of a service's functionality, a detailed documentation of the interfaces as provided for the services created during the MORE implementation phase and documented in D4.4 is necessary.

To enable easy creation of the human readable documentation, a WSDL documentation generator has been created by PRO DV as part of the MORE utilities framework. The generator is released as open source in parallel to the main parts of the middleware framework.

The `wSDL2rtf`-tool extracts for a service every port-definition with the associated operations and their messages (in, out and faults). It also looks for the documentation tags, which belongs to every element

- Service
- Porttype
- Input
- Output
- fault.

While extracting it saves these informations in an own data format (see under `wSDL2rtf.model`). This data format is later transformed into the documentation. The next steps describe how to prepare the `.wsdl`-file and how to use this tool to successfully create a `wSDL`-documentation.

### Step 1 - The `.wsdl`-file

When you write / create your `.wsdl`-file you have to use the additional `<wsdl:documentation>` .. `</wsdl:documentation>` tag to add your desired documentation. You have to add these tags to the following fields:

- `wsdl:service`
- `wsdl:porttype`
- `wsdl:operation`
- `wsdl:input`
- `wsdl:output`
- `wsdl:fault`.

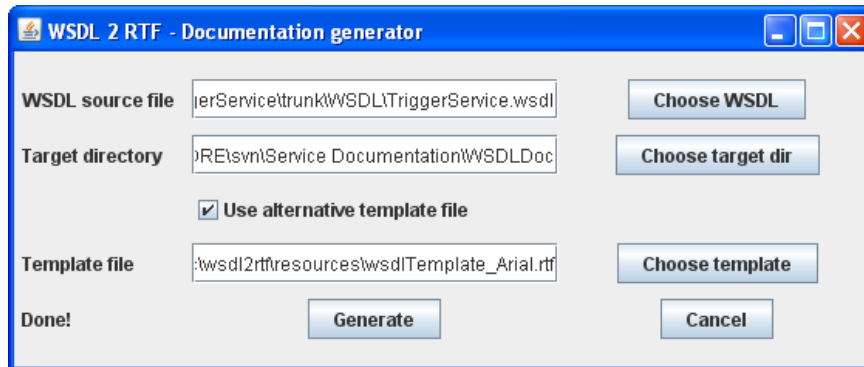
All other documentation will be ignored.

### Step 2 - Calling the tool

You can use the tool by two ways, commandline or file-chooser-dialogs. When you choose to use commandline, you have to specify two mandatory and one optional parameter:

- `SourceWSDL`: the complete path and filename of the `.wsdl`-file to parse
- `TargetDirectory`: the path to the target directory where generated `.rtf`-documents were saved
- `templateFile`: (additionally) the used `.rtf`-template file.

This way is recommended when you want to generate documentation of various `.wsdl`-files (eg. in a batch-work). When you choose to use the file-chooser-dialogs you just have to call the tool with the parameter `--gui`. It will now open a simple gui. Here you can choose the `.wsdl`-file to generate from, the target directory. And, if you want, an alternative template-file. At the end both of ways the tool reads the source file, extracts desired information and generates the `.rtf`-document.



**Figure 5: MORE WSDL 2 RTF generator GUI**

File structure of the WSDL documentation generator project:

```

|
+ bin
| |-- (class files)
+ doc
| |-- README.txt (Usage information)
| |-- HOWTO.txt (Template information)
+ license
| |-- License.GNU.LGPL
+ lib
| |-- rtftemplate-1.0.1-b14.jar
| |-- velocity-dep-1.5.jar
| |-- jdom.jar
| |-- oro.jar
| |-- spring-core.jar
| |-- spring.jar
| |-- freemarker.jar
| |-- logkit-1.0.1.jar
| |-- commons-beanutils.jar
| |-- commons-collections.jar
| |-- commons-digester-1.7.jar
| |-- commons-io-1.1.jar
| |-- commons-logging.jar
+ src (lists only necessary packages for developers)
| |-- wsdl2rtf.Main.java
| |-- wsdl2rtf.gui
| |-- wsdl2rtf.model
| |-- wsdl2rtf.resources
| | |-- wsdlTemplate.rtf
| | |-- wsdlTemplate_Arial.rtf
| | |-- WSDLFields.xml
| | |-- RTFTemplate.dot

```

The WSDL documentation generator is based on a rtf templating engine, which is based itself on the generic templating engine “velocity”. The generator parses the input WSDL and extract

the necessary fields as input for the rtf generation engine. The following steps are needed to generate the documentation:

- design your rtf model with MS Word by using merge fields (MERGEFIELD), hyperlink fields (HYPERLINK) and bookmarks (BOOKMARK, to manage start/end loop). Standard templates are provided by MORE in the resources folder as documented above.
- prepare your context with JAVA objects. This is done by the generator and does not need any user interaction
- merge your rtf model (Template) with your JAVA objects (Context) by using RTFTemplate. This step is automatically performed when the generation process is started.

Example of how a rft template has to look:

The fields are filled in following the WSDL structure. Macros have to be enabled in Word for the final step of merging.

«\$PortType.Documentation»
<b>Operation «\$PortType.Operation.Name»</b>
«\$PortType.Operation.Documentation»
<b>Messages</b>
<b>Input</b> «\$PortType.Operation.InMessages.Name» «\$PortType.Operation.InMessages.Documentation»

Component	Software used	Context	License
WSDL2RTF	RTF Template: <a href="http://rtftemplate.sourceforge.net/">http://rtftemplate.sourceforge.net/</a>	Java rtf templating engine	GNU LGPL
	Spring framework: <a href="http://www.springframework.org/">http://www.springframework.org/</a>	Centralized, automated configuration and wiring of application objects.	Apache license
	Velocity : <a href="http://velocity.apache.org/engine/releases/velocity-1.5/">http://velocity.apache.org/engine/releases/velocity-1.5/</a>	Apache Velocity is a general purpose template engine.	Apache license

	Freemarker : <a href="http://freemarker.sourceforge.net/">http://freemarker.sourceforge.net/</a>	FreeMarker is a generic tool to generate text output based on templates.	BSD license
--	---	--	-------------

## 4. MIDDLEWARE COMPONENTS DEVELOPERS GUIDE

### 4.1. Core Management

The MORE Core Management component is responsible for starting, stopping, managing the different MORE services present on a given node. Respecting the modular approach adopted in MORE, the Core Management component is provided as an OSGi bundle ready to be deployed. As the basic component of the MORE middleware, it is the only mandatory component in order to deploy MORE services. For that purpose, it has been designed to be as simple as possible, embedding only the necessary features necessary for modular middleware components and MORE Services.

#### 4.1.1. Component design details

To summarize the Core Management architecture, here are the main java classes used:

The screenshot shows a web browser window displaying the 'Class List' for the 'eu.more.core' package. The browser address bar shows 'D:\Documents And Settings\716820\Bureau\Doc\html\index.html'. The page content includes a navigation menu on the left with options like 'Class List', 'Class Hierarchy', 'Class Members', 'Graphical Class Hierarchy', 'Package List', and 'File List'. The main content area is titled 'Class List' and contains a list of classes and interfaces with brief descriptions. The classes listed are:

- org.osgi.service.cm.Configuration
- org.osgi.service.cm.ConfigurationAdmin
- org.osgi.service.cm.ConfigurationEvent
- org.osgi.service.cm.ConfigurationException
- org.osgi.service.cm.ConfigurationListener
- org.osgi.service.cm.ConfigurationPermission
- org.osgi.service.cm.ConfigurationPermissionCollection
- org.osgi.service.cm.ConfigurationPlugin
- eu.more.core.internal.Controller
- eu.more.core.internal.CoreActivator
- eu.more.core.internal.CoreServer
- eu.more.core.internal.DirectoryWatcher
- org.osgi.service.cm.ManagedService
- org.osgi.service.cm.ManagedServiceFactory
- eu.more.core.internal.MOREService
- eu.more.core.internal.ResourceThreadFactory

At the bottom of the page, it says 'Generated on Fri Nov 28 11:35:15 2008 for eu.more.core by doxygen 1.5.7.1'.

*eu.more.core.internal.controller* : in charge of adding/removing services from the CoreServer  
*eu.more.core.internal.DirectoryWatcher*: responsible for probing "load" directory content and deploy all services found in it.

*eu.more.core.internal.CoreActivator*: Start the local server which host the DPWS devices, creates a devices hosting the MORE Services, and register services where they are found locally.

[eu.more.core.CoreServer](#): responsible for creating a new DPWS server, and is able to add services to it.

The design and current features of the Core Management Service has been voluntarily kept very simple, in order for it to run on limited capacity devices.

#### 4.1.2. Future Developments

Next step in implementation is to further develop current Core Management to make it a MORE Service available on the network the way middleware component are already running. Indeed to be compliant with the D2.1, the Core Management Service have to provide some more features currently absent from the implementation like remote management of MORE services or capability advertising. Some more guidance will be provided in the next release of this Software Prototype and Integration.

#### 4.1.3. Generated Javadoc

The generated Javadoc for the CMS is available on the MORE website: [www.ist-more.org](http://www.ist-more.org)

## 4.2. Group Management

### 4.2.1. Component architecture/design details

The Group Management Service (GMS) is responsible for the administration and management of groups and their members within the MORE Middleware Platform. The Group Management Service provides a decentralised point of contact for MORE services to create, delete, modify, query a specific group, and to send messages to a group.

The logic which comes from the policies is contained within the GMS itself for the latest drop of software (end of 2<sup>nd</sup> year of the project). The next software drop of the project will include a Policy Engine Service (see Service with dotted lines in Figure 1 below) and the logic currently in GMS will be migrated into policies and processed with this new Service. During the implementation phase a decision was made to decouple the policy processing from GMS in order to make the system more modular. Additionally the functionality of the Group Communication Service (GCS) is now embedded into GMS in order to simplify the group communication protocol flow for performance considerations.

In order to simplify the usage of GMS for the developer we have created a GMS client. The developer is still free to develop using direct invocation on the GMS. The client helps to hide the constraints that the rules in the GMS Tutorial enforces (User Guide).

The GMSClient has one constructor-taking one argument- and one static method to return the GMSClient. If the constructor is used, the client will look for the GMS service using the deviceExplorer. This will instantiate the GMSInvoker. The argument is for the serial number of the device that the GMS resides on. If the static method is used, then a GMSClient is returned with a GMSInvoker instantiated with a GMS service that is LOCALLY found on the system. If the GMS is not on the local machine, then this static method will throw a "GMSNotFoundException". Possibly a good rule of thumb is to try the static method and if the exception is thrown to use the constructor to instantiate it.

When the GMSClient is instantiated, the developer is free to use the methods contained in the GMSClient to invoke on the GMS.

### Member Interaction in the GMS

For member interaction in the GMS, it was decided to use the deviceId of the Device that the service is located on, with the serviceType of the service. Given that a member is defined as a string, a string must be built to identify the service from the GMS.

Therefore follow this format always when using members with the GMS:

```
String newServiceDetails = "";

AddMemberToGroup newGroupService =
fact.createAddMemberToGroup();

newGroupService.setGroupName(groupName);

newServiceDetails += deviceId + "*";
newServiceDetails += serviceType.getNamespaceURI() + "*";
newServiceDetails += serviceType.getLocalPart();

newGroupService.getMember().add(newServiceDetails);

gmsInvoker.invokeaddMemberToGroup(newGroupService);
```

This will ensure that the GMS gets the correct information when the string is tokenized.

This approach should also be used when interacting with the GMS (deleteMemberFromGroup, createGroup, addMemberToGroup, queryGroup)

#### 4.2.2. Motivation for this particular design

The reason this particular design was chosen is because it allows for the dynamic creation, deletion and reconfiguration of groups within the MORE middleware. This attribute is beneficial as it provides the capability to create for example a patient monitoring group. This group could initially consist of a patient and a nurse. If the patient's vitals exceed a predefined threshold a doctor can be automatically added to this group.

By implementing a Policy Engine Service this is to allow the policy engine to make decisions based on data sent to it from the GMS. The policy engine should make decisions based on the policy rules available to it at the time it receives the data from the GMS. The policy engine is the decision maker for the GMS. The policy engine needs to use JAVA classes to be compatible with the core used for the MORE project. This paper deals with the implementation of this policy engine using the Groovy programming language. This approach sees the flexibility of Groovy being exploited to allow ease of use for the developer.

#### 4.2.3. Future Developments



Future Development includes the extension of the Policy Engine to be fully functional for the purposes of the MORE project. Currently it has only limited capability as a direct result of its complexity and the limited timeframe available for its development. However there is ample documentation for the Policy Engine available on the MORE website and as it stands it adaptable to any application.

#### 4.2.4. Generated Javadoc

The following diagram shows the index.html with just an outline view, for further details please see the MORE website.

The screenshot shows a Javadoc overview page for GMS classes. On the left, there is a navigation pane with 'All Classes' and 'Packages' sections. The 'All Classes' section lists various classes such as `AddedToGroupNotificationF`, `AddMemberToGroup`, `AddMemberToGroupImpl`, `AddMemberToGroupRespc`, `AddQueryMemberToGroup`, `AlreadyAddedException`, `CreateGroup`, `CreateGroupImpl`, and `CreateGroupResponse`. The 'Packages' section lists `eu.more.gms`, `eu.more.gms.Exceptions`, and `eu.more.gms.generated`. The main content area displays a table of packages with the following entries:

<a href="#">eu.more.gms</a>	
<a href="#">eu.more.gms.Exceptions</a>	
<a href="#">eu.more.gms.generated</a>	
<a href="#">eu.more.gms.generated.jaxb</a>	
<a href="#">eu.more.gms.generated.jaxb.impl</a>	
<a href="#">eu.more.gms.testing</a>	
<a href="#">gms.Bundle</a>	
<a href="#">gmsClientTest</a>	
<a href="#">testing.eu.more.gms</a>	

The page also features navigation tabs for Overview, Package, Class, Tree, Deprecated, Index, and Help, along with 'PREV NEXT' and 'FRAMES NO FRAMES' options.

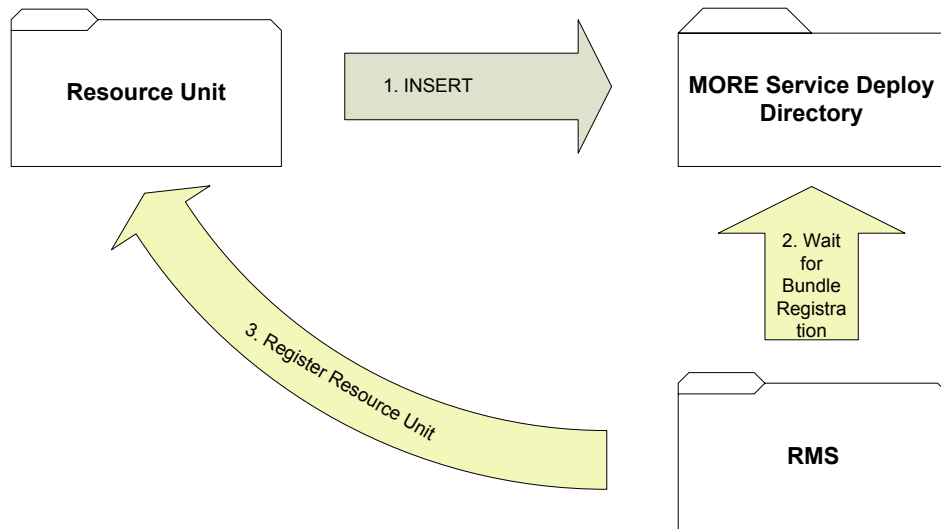
Figure 6: GMS Classes overview

## 4.3. Resource Management

### 4.3.1. Component architecture/design details

The startup workflow of the Resource Management Service after writing a Resource Unit comprises three steps. The first step is to add the Resource Unit to the MORE Service Deploy Directory. When the Resource Management Service was started by the MORE core it automatically waits for Resource Units that publish their services via OSGI (Step 2). The third step is the registration of the methods and resources which are published by the Resource

Units. These operations are then accessible via the DPWS connectors from inside and outside the MORE device.



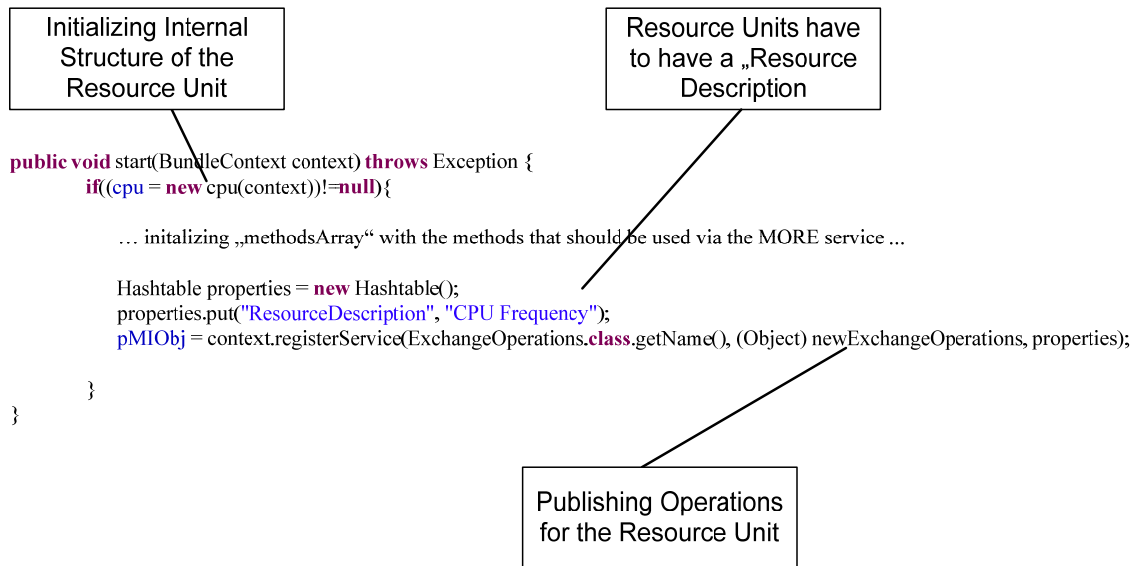
**Figure 7: Workflow Resource Unit Deployment**

#### 4.3.2. Motivation for this particular design (Optional)

The plug-in structure for the RMS was chosen to enable service developer and application designer who uses the MORE middleware and the RMS to adapt the RMS to their needs.

#### 4.3.3. Future Developments

Extending the Resource Management Service is quite easy. Because of the generic interface structure the interface keeps the same all the time and the service developer can focus on writing his Resource Unit(s). A Resource Unit comprises the methods that can be invoked to use a resource and it publishes OSGI services so that the Resource Management Service can register the Resource Unit to itself.



**Figure 8: Resource Unit Initializing**

The starting point for a service developer is to generate an OSGI bundle that provides the functionality of the Resource Unit. After creating the activator of the Resource Unit must be changed so that the Resource Management Service can access it. Because the Resource Unit itself is no DWPS service it must be accessed via OSGI. The OSGI service which has the type *ExchangeOperations* that have to be published by the Resource Unit. The *ExchangeOperations* class comprises two members:

1. An array of *java.lang.reflect.Method* which provides all operations of the Resource Unit that should be used via the Resource Management Service.
2. One *java.lang.Object* which have to be used to invoke the operations of the resource unit.

#### 4.3.4. Generated Javadoc

The Resource Management Service is documented completely with doxygen. The documentation can be found on the MORE website. The overview is provided in Figure 9.

**Resource Management Service**

- Class List
- Class Hierarchy
- Class Members
- Package List
- File List

Main Page Packages Classes Files

Packages

### Package List

Here are the packages with brief descriptions (if available):

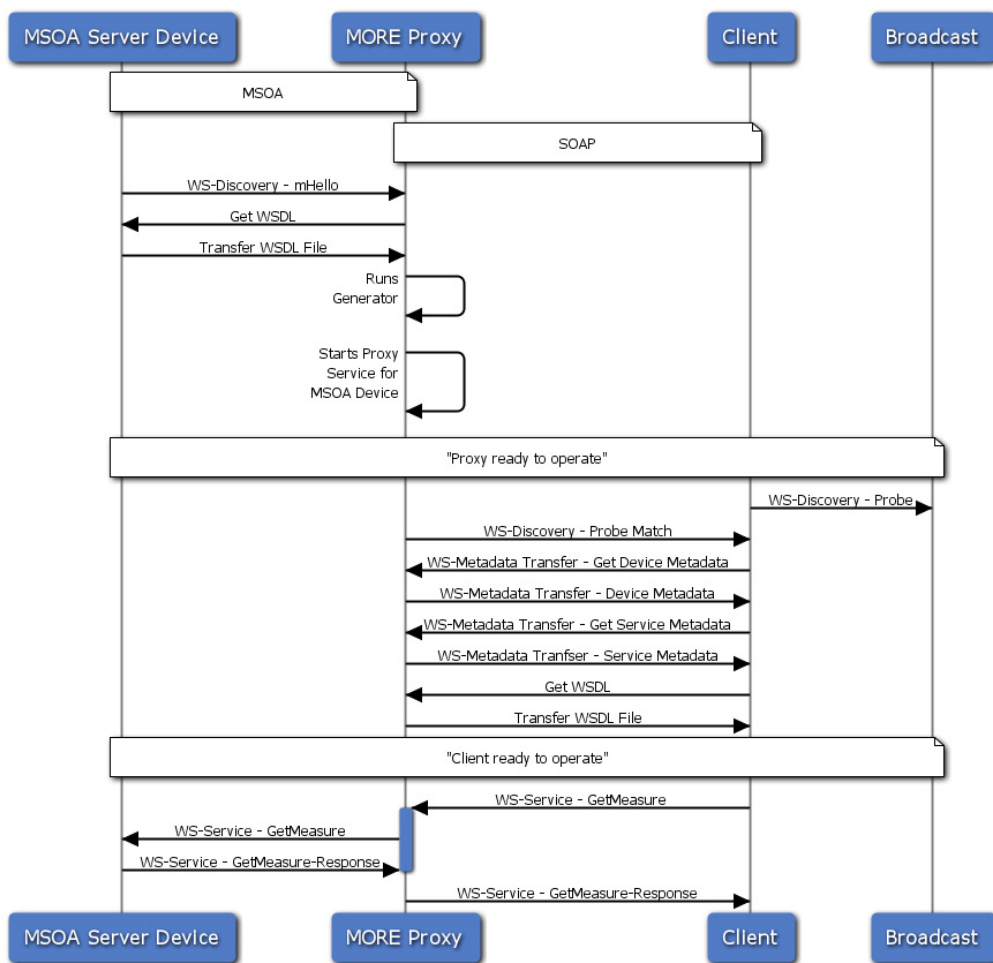
eu	
eu.more	
eu.more.ResourceManagementService	
eu.more.ResourceManagementService.generated	
eu.more.ResourceManagementService.generated.jaxb	
eu.more.ResourceManagementService.generated.jaxb.impl	

**Figure 9: Packages RMS**

## 4.4. $\mu$ SOA Proxy Service

### 4.4.1. Component architecture/design details

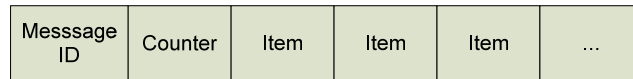
The implementation of the  $\mu$ SOA protocol requires a  $\mu$ SOA Proxy Server instance for translating SOAP to  $\mu$ SOA messages and vice versa. Processing of  $\mu$ SOA messages is handled in the Core Management Service. Other clients are not able to decode the content from the  $\mu$ SOA message since they utilize SOAP for DPWS and Web Service compliancy. Since the Proxy Service handles the translation between  $\mu$ SOA and SOAP messages and vice versa the Proxy Service is a mandatory requirement for DPWS client applications.



**Figure 10: Sequence Chart for  $\mu$ SOA Communication**

The encoding scheme for the  $\mu$ SOA protocol is also generated during the start of the service on the  $\mu$ SOA Proxy Service. The scheme is generated with the help of the WSDL description and describes all objects of the  $\mu$ SOA message in the same order as they are defined in the WSDL description. Since the WSDL definition is transferred from the MORE Server Device to the  $\mu$ SOA Proxy Service before starting the proxy instance of the service,

the WSDL description is always consistent on both devices and allows for a consistent encoding of messages. The  $\mu$ SOA encoding scheme replaces the serialization of the Java objects into XML-Streams, dissolves the objects into generic data types, and encodes them as byte-streams for efficiency.



**Figure 11:  $\mu$ SOA Message Format**

A counter announces the number of elements of a list in order to allow a client to resolve the number of elements within the list. The encoding scheme is limited to the ASCII-Character set and does not allow e.g. UTF-8 characters due to performance considerations. The encoding scheme considers all known primitive data types (known sizes). The WSDL describes each complex data type based on primitive data types which are all processed by the  $\mu$ SOA encoding. One exception is the data type: String, whose length is unknown. A special separator symbol („|“) has been defined in order to separate the string from following data types within the message. Due to this property, the “|” symbol is not allowed as content of a string within  $\mu$ SOA messages:

Dez	Hex	Okt	ASCII
123	0x7B	173	

**Figure 12: Separation Code for Strings**

The runtime integration of the proxy service are supported through other enhancements of the Core Management Service. Incoming  $\mu$ SOA messages have to be identified and redirected towards the  $\mu$ SOA decoding instead of the standard XML parser for typical SOAP messages. Additionally, outgoing messages are identified through internal mappings of the service, encoded into the  $\mu$ SOA format and directly send to the receiver.

#### 4.4.2. Future Developments

The implementation of the  $\mu$ SOA proxy service is nearly finished. A complete test for scenario 1 has been performed successfully. Some automation and management functionality for convenient set up and configuration of the proxy service are still missing. These measures will simplify the handling of the proxy service and will have an influence on the How To Use section of this document.

Additionally we are currently working on a J2ME CLDC 1.1 compliant  $\mu$ SOA service endpoint which can be conveniently integrated into the scenario.

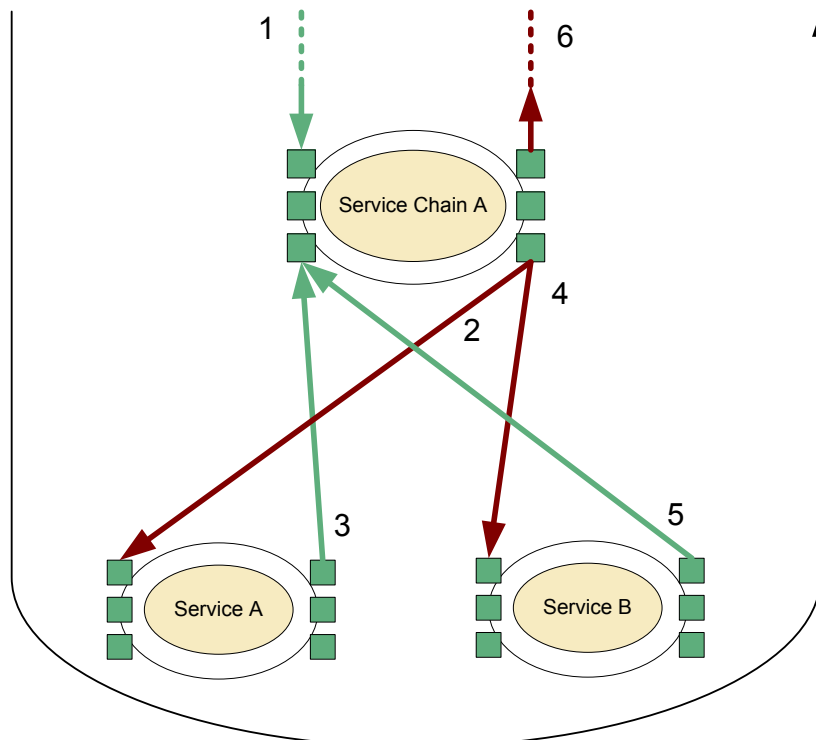
#### 4.4.3. Generated Javadoc

Please find on the MORE website, the complete javadoc documentation. It is also available with the  $\mu$ SOA software package in the doc folder.

## 4.5. Service Chaining

### 4.5.1. Component architecture/design details

Creating a new MORE service chain which combines several MORE services to a new MORE service is quite simple for a MORE service developer. Instead of creating a new MORE service a MORE service developer can combine MORE services with a Service Chain XML Configuration File (SCC). The functionality of the MORE service chain mechanism adjusts the specification D2.1 to the functionality provided by the MORE Core. Invoking a MORE service chain works as depicted in Figure 13.



**Figure 13: MORE Service Chain Dataflow**

Service Chain A is invoked by a MORE client in the network or local. Then the dependent MORE services are sequentially invoked by Service Chain A. The conversion of the different namespaces is done, if needed, by Service Chain A.

### Creating a Service Chain XML Configuration File (SCC)

Creating a MORE service chain starts with the creation of a SCC. The SCC is an XML is the following schema:

```
<xsd:schema targetNamespace="ServiceChaining">
  <xsd:element name="ServiceChainObject">
    <xsd:complexType>
      <xsd:sequence maxOccurs="1" minOccurs="1">
        <xsd:element name="ActiveInterval" type="xsd:int" maxOccurs="1" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

<xsd:element name="ServiceChainIdentifier" type="xsd:string"/>
<xsd:element name="NameSpace" type="xsd:string"/>
<xsd:element name="UID" type="xsd:string"/>
<xsd:element name="Operation" type="xsd:string"/>
<xsd:element name="DependentServices">
  <xsd:complexType>
    <xsd:sequence maxOccurs="unbounded" minOccurs="1">
      <xsd:element name="DependentService">
        <xsd:complexType>
          <xsd:sequence maxOccurs="1" minOccurs="1">
            <xsd:element name="ServiceOperation" type="xsd:string"/>
            <xsd:element name="NameSpace" type="xsd:string"/>
            <xsd:element name="Binding" type="xsd:boolean"/>
          </xsd:sequence>
          <xsd:attribute name="ServiceIdentifier" type="xsd:string"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="WSDL" type="xsd:string"/>
</xsd:sequence>
<xsd:attribute name="Active" type="xsd:boolean"/>
</xsd:complexType>
</xsd:element>
</xsd:schema>

```

**Figure 14: SCC XML Schema**

The "ServiceChainIdentifier" is the name and address of the service chain. The "NameSpace" is the namespace of the service chain and "UID" is the unique identifier of the physical endpoint of the service chain. "Operation" is the name of the operation of the service chain. Between the WSDL tags one has to enter the WSDL which can be used to access the service chain. One of the most important elements in the XML is the "DependentServices" -Element where one has to specify the services which are in the service chain. The order of the services in the XML file is the invocation order of the services in the chain. When you specify the service chain you must take into account that the output XML element of a service must have the same XML schema like the input XML element of the successor service. The only exception is the namespace because it can be different. An example SCC can found in Figure 15.

```

<?xml version="1.0" encoding="UTF-8" ?>
<tns:ServiceChainObject xmlns:tns="ServiceChaining" Active="false">
  <ServiceChainIdentifier>ServiceChainA</ServiceChainIdentifier>
  <NameSpace>http://www.ist-more.org/ServiceChaining</NameSpace>
  <UID>646961683513734323676644</UID>
  <Operation>GetMeasure</Operation>
  <DependentServices>
    <DependentService ServiceIdentifier="http://www.ist-more.org/LocationService">
      <ServiceOperation>GetLocation</ServiceOperation>
      <NameSpace>http://www.ist-more.org/LocationService</NameSpace>
      <Binding>true</Binding>
    </DependentService>
    <DependentService ServiceIdentifier="http://www.ist-more.org/MeasurementService">

```

```

        <ServiceOperation>GetMeasureWithLocal</ServiceOperation>
        <NameSpace>http://www.ist-more.org/MeasurementService</NameSpace>
        <Binding>true</Binding>
    </DependentService>
</DependentServices>
<WSDL><![CDATA[“PLEASE FILL IN WSDL HERE”]]></WSDL>
</tns:ServiceChainObject>
    
```

**Figure 15: SCC Example**

#### 4.5.2. Future Developments

The next step for MORE service chaining is to add the support of remote MORE services which means that the dependent services of the MORE service chain are not on the same MORE device but accessible with a network connector on a remote MORE node.

#### 4.5.3. Generated Javadoc

Service Chaining is documented completely with doxygen. The documentation can be found on the MORE website. The overview is provided in Figure 16.

**Figure 16: Packages Service Chain**

## 4.6. Communication Priorization Service

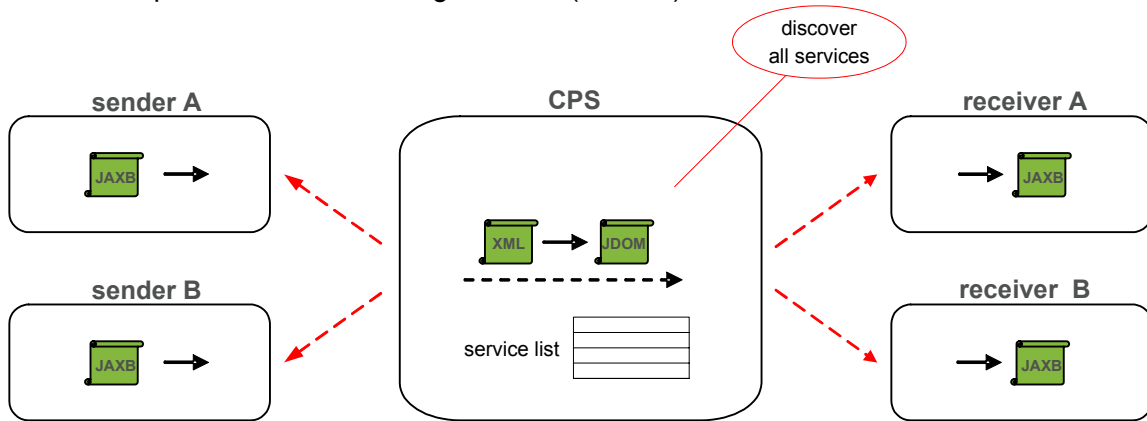
#### 4.6.1. Processing of Messages

In all application scenarios the Communication Priorization Service (CPS) is utilized as intermediate service and is not the primary target service for a message exchange. The DPWS4J stack utilizes developer-friendly but unflexible JAXB XML bindings which define a precise interface for incoming messages. This is not applicable for the CPS since its incoming message interface has to be generic and usable without any knowledge of the services it is communicating with. Hence the CPS does not utilize the JAXB XML binding but a transparent

message input. Instead of forwarding a message towards its input method by mapping its JAXB bindings, the XML stream is directly send to the input method. In order to enqueue the message and to forward it towards its ultimate receiver, the message is internally processed by the JDOM XML binding. The receiver obtains the message as ordinary XML stream and is can process it as usual with the service specific JAXB XML binding (see fig 5 und 6).

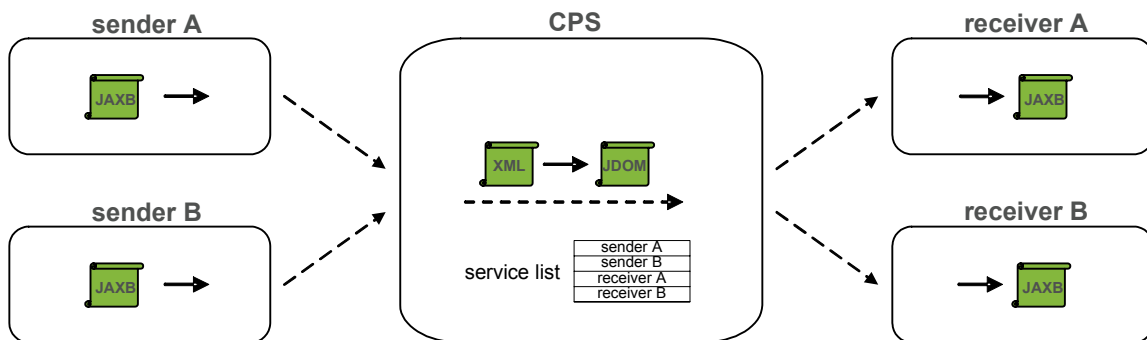
#### 4.6.2. Sequence of Operations

During the start of the CPS all available devices and their hosted services are discovered through a multicast probe (see fig 6). The service information is then stored in a service list in order to allow forwarding of incoming messages to their ultimate receiver based on additional information provided in the message header (see 1.5).



**Figure 17: CPS Service Discovery**

As soon as all information is read from the header, the message can be enqueued. The dequeuing depends on the utilized prioritization algorithm as previously described (except for the MORE Virtual Clock Algorithm). Before sending out the message, its ultimate receiver (service endpoint) is determined through the Service ID from within the service list.



**Figure 18: Simple CPS Scenario**

#### 4.6.3. Future Developments

The current version of the CPS is working as described. All three algorithms are implemented, as well as the transparent message input and the message forwarding mechanism. The algorithms themselves are currently under simulative evaluation for

performance considerations in WP530. The information gathered in WP530 will be fed back into the implementation in order to increase CPS performance. Some further code optimizations are also planned for the near future.

## 5. MORE SERVICES DEVELOPERS GUIDE

### 5.1. Measurement Services

The Measurement Services specified in Deliverable D2.1 are abstract services and provide the generic frame for implementations. Here it will be demonstrated how new implementations for different sensors can be integrated.

#### 5.1.1. Component architecture/design details

The generic Measurement Service is completely based on Java, providing system independency. For development purposes in parallel to the source code according Eclipse development environment configuration files are available, enabling import of the base project into the IDE. No further setup is needed.

For some specific services additional libraries and operating system dependencies exist, i.e. for serial communication native libraries are needed.

Finished measurement services: Virtual gas measurement service, virtual blood glucose value measurement service, virtual luminescence sensor, gas measurement service (G750 Polytektor II), blood glucose level monitors (Roche AccuChek, smartLab global)

Work in progress measurement services: Gateway JXTA measurement service, DataTaker DT80 data logger measurement service, multi-service measurement service.

#### 5.1.2. Future Developments

New measurement devices can be easily integrated into the service without deeper understanding of the interworkings of the MORE middleware. This section documents with an example how to develop access to new sensor devices.

All measurement services have to implement the `eu.more.measurement.service.datareaders.DataReader` interface. The short interface summary below documents the current work in progress measurement service with different access methods for time interval handling of historical data. The most minimalistic measurement service only has to implement the `readData(Position[] position)` method.

Method Summary	
<a href="#">Measurement</a>	<code>readData(java.util.Calendar dataFrom, java.util.Calendar dateTill)</code> Handles sensor device specific reading and interpreting of measured values.
<a href="#">Measurement</a>	<code>readData(java.util.Calendar dataFrom, java.util.Calendar dateTill, java.util.List&lt;Area&gt; areasOfInterest)</code> Handles sensor device specific reading and interpreting of measured values.
<a href="#">Measurement</a>	<code>readData(java.util.Calendar dataFrom, java.util.Calendar dateTill, Position[] position)</code> Handles sensor device specific reading and interpreting of measured values.
<a href="#">Measurement</a>	<code>readData(java.util.List&lt;Area&gt; areasOfInterest)</code> Handles sensor device specific reading and interpreting of measured values.
<a href="#">Measurement</a>	<code>readData(Position[] position)</code> Handles sensor device specific reading and interpreting of measured values.

Two different options exist for handling data passing to the MORE middleware stack: XML based responses as also described for the user's guide on receiving and interpreting the middleware messages. The second option utilizes automatic (un-)marshalling of the specific objects, enabling convenient use of plain Java objects. The example below demonstrates how easy it is to pass new values to the middleware.

```
package eu.more.measurementservice.datareaders;

import eu.more.measurementservice.generated.jaxb.KeyValuePair;
import eu.more.measurementservice.generated.jaxb.Measurement;
import eu.more.measurementservice.generated.jaxb.Position;

public class MySensorReader implements DataReader{

    // Method is called when a new measurement is requested or the time
    // interval for two succeeding measurements for subscriptions has
    // passed.
    @Override
    public Measurement readData(Double[] position) {

        // Create a new measurement
        Measurement myMeasurement = new
eu.more.measurementservice.generated.jaxb.impl.MeasurementImpl();
        // Set the name of the sensor node.
        // This enables differentiation of measurements from
        // different sensors provided by a single service
        myMeasurement.setNodename("My Sensor");
        myMeasurement.setTime(new java.util.GregorianCalendar());
        // Set the position of this sensor
        // Usually the position provided in position should be
used,
        // as this is provided by a local LocationService
        Position pos = new
eu.more.measurementservice.generated.jaxb.impl.PositionImpl();
        // Override position with a fixed position
        // (here in Dortmund, Germany)
        pos.setLatitude(51.492481);
        pos.setLongitude(7.402202);
        // Prepare a new measurement associated with a value
        KeyValuePair kvp = new
eu.more.measurementservice.generated.jaxb.impl.KeyValuePairImpl();
        // Set the value name and the value itself
        // Here should go your code on retrieving
        //the real sensor value, e.g. by reading data from
        // the serial interface.
        kvp.setKey("MyValue");
        kvp.setValue("321");

        // Add position to measurement
        myMeasurement.setPosition(pos);
        // Add value(s) to measurement. In case several
```

## 5.2. Glucose Send Data Service

### 5.2.1. Component architecture/design details

This Service is implemented in tn JAVA using the standard MORE development methods and tools. The final version of this service is provided as LGPL, except the component implementing the sensor readout stack, which is proprietary to the provider, Applied Logic Laboratory. At the point of the creation of this documentation this service is in beta version. The main difference between the current version and the final version will be the level of integration of the security features.

This service is implemented in three components. This three layer architecture is necessary to enable the control of the measured data to be exercised by the patient, because measurement errors are possible and could have serious consequences. Moreover, the OSGI service architecture and the chosen architecture enable the very easy integration of new Blood Glucose sensors. The following Figure 19 : Three layer architecture of the device services in the Health Care Scenario demonstrates the three layer architecture:

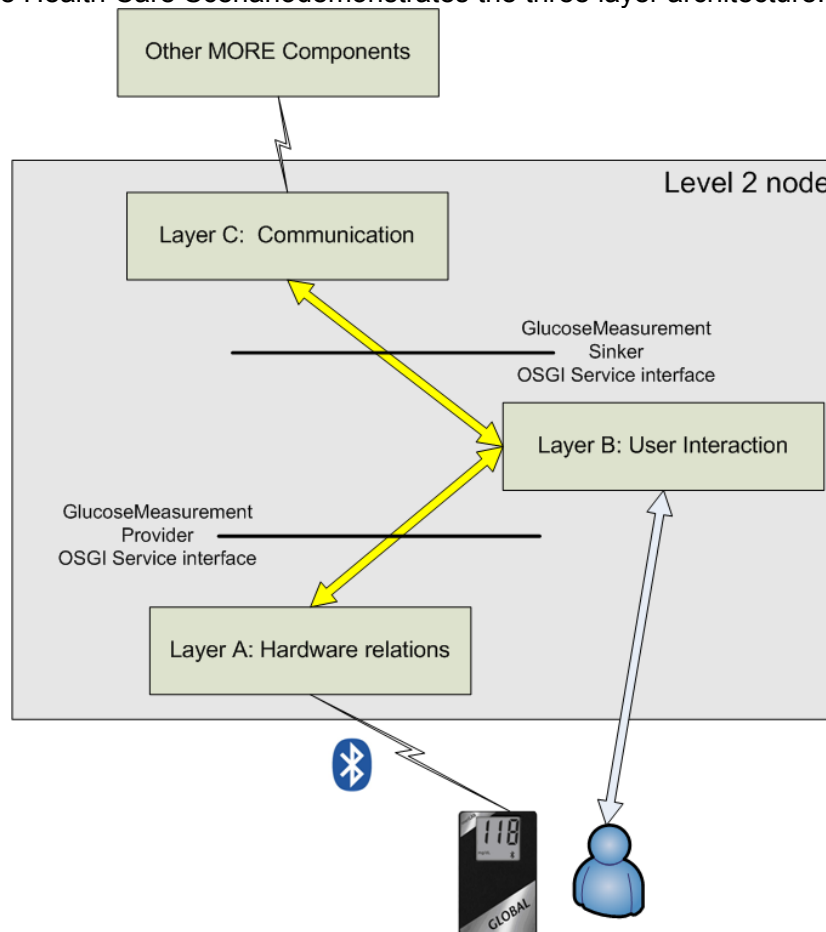


Figure 19 : Three layer architecture of the device services in the Health Care Scenario

Layer A is responsible of the readout of the sensor. Layer B is the application layer which is able to transfer the measured data from layer A to layer C. Layer C realizes the MORE message driven interface and able to publish the measured data. The following paragraphs describe each layer more specifically.

### Layer A

Layer A is on the hardware related technical level. On the hardware level, it is implementing the Bluetooth communication protocol of the SmartLab Global Blood Glucose sensor, and thus enable the read out of a new blood glucose measurement value from the sensor.

On the JAVA level, the component is implementing the following simple JAVA interface:

```
public interface GlucoseMeasurementProvider {
    public interface Listener {
        public void newGlucoseMeasurement(GlucoseMeasurement
measurement);
    }
    public void registerGlucoseMeasurementListener(Listener listener);
    public void deRegisterListener(Listener listener);
}
```

The interface enables the registration and de-registration of an appropriate listener object, and it is expected that any implementation will call the registered listener's newGlucoseMeasurement method whenever a new glucose measurement is available. The implementation is also expected to register itself as a GlucoseMeasurementProvider OSGI service to the OSGI environment.

### Layer B

Layer B is the layer of the user control. It is assumed to register itself as a listener to all available implementations of layer A. Once a new glucose measurement is available, it shows it on the user interface and waits for the acknowledgment of the user. Once the acknowledgment is received, it tags the value with the data of the user and propagates the value to layer C and handles any reply. On the Java level, it implements both the GlucoseMeasurementProvider.Listener and the GlucoseMeasurementSinkers.GlucoseMeasurementReplyListener interfaces.

### Layer C

Layer C is the communication layer. On the java level, it implements the following interface:

```
public interface GlucoseMeasurementSinkers {
    public interface GlucoseMeasurementReplyListener {
        public void replyForMeasurement(GlucoseMeasurement measurement,
int replyCode);
    }
    public void newGlucoseMeasurements(User user, Collection
measurements,
GlucoseMeasurementReplyListener replyListener)
throws GlucoseMeasurementException, IOException,
InsufficientPrivilegesException;
}
```

On the Web-Service level, it implements a MORE service with the Web-Service event of the new measurement value. It also provides an operation through which the last measurement value can be accessed.



### 5.2.2. Motivation for this particular design

The flexibility of the SOA design together with the flexibility of the realized OSGI concept in the MORE Core implementation is not restrictive on the architecture of the application layer. This particular three layer architecture was chosen because it has two very important advantages:

- Inclusion of new type of blood glucose sensors is easy and transparent. A new OSGI service implementing the interface of Layer A and able to read out the sensor has to be implemented and deployed; the components of Layer B automatically detects the presence of this new implementation and start using it.
- In certain circumstances it is possible that the patient control of the data is not necessary. In this case only the application level components of layer B has to be modified and not layer A and Layer C.
- Using this type of it is also flexible how the read out data are published. In this case only layer C has to be modified on the patient's node.

### 5.2.3. Future Developments

As mentioned earlier, the final version of this service will integrate the security features in a seamless manner.

If a new type of sensor is needed to be enabled by this service, an OSGI bundle providing and advertising the *GlucoseMeasurementProvider* interface is needed. This will be automatically detected by the rest of the service implementation and made use of.

The following code illustrates how from your bundle activator you can advertise your implementation of *GlucoseMeasurementProvider*:

```
myProvider          =          new          MyProvider();          //implements
GlucoseMeasurementProvider
Properties properties = new Properties();
context.registerService(
hu.all.more.diaball.interfaces.GlucoseMeasurementProvider.class
.getName(), myProvider, properties);
```

## 5.3. Local Storage Service

### 5.3.1. Component architecture/design details

This service is implemented by Applied Logic Laboratory in JAVA 1.5, as reference implementation of the Local Storage Service set out in D2.1. This implementation is licensed as LGPL.

The source of this service can be found under `hu.all.more.localstorage`. To run or compile this service you need to have Java version at least 1.5.



To run the service you need to:

1. Copy `Projectfiles/Runtime/Run/LocStorage.prt` into a directory called `LocalStorage_prt` under the root directory of your started JVM.
2. Edit it if you want to change the name of the directory where local storage is supposed to store its data.

This implementation provides the permanent storage in the file system. Each folder on the service interface corresponds to a folder in the actual file system.

The `hu.all.more.localstorage` package itself contain the activator of the bundle and the service implementation.

The class `hu.all.more.localstorage.verifier.LocalStorVerify` can be used to check if your parameters you are going to provide to the local storage service are correct. The `isLength()` method tells if the supplied folder name can be excepted by its length. The `isValid()` method can be used to check if the supplied parameter can be a valid folder or data key name.

The wsdl directory contains the wsdl file defining the interface of the service.

The classes and packages under `hu.all.more.localstorage.generated` are generated from the wsdl automatically by the source code generator tool.

#### 5.3.2. Future Developments

Future developments of this service cold be an implementation using SQL or other database engine. However, the current implementation can not be used as a base for that development, except the wsdl file and the generated classes.

## 5.4. Compression Service

#### 5.4.1. Component architecture/design details

This service is implemented by Applied Logic Laboratory in JAVA 1.5, as reference implementation of the Compression Service set out in D2.1. This implementation is licensed as LGPL.

The source of this service can be found in the source repository as `CompressionService`. To run or compile this service you need to have Java version at least 1.5.

To run the service itself you need the either the precompiled osgi bundle for it your need to compile your own version of source files.

The current implementation implements only one compression algorithm, called "ZIP" and based on the standard java zip file generator classes.

The `hu.all.more.compressionervice` package itself contain the activator of the bundle and the service implementation.

The wsdl directory contains the wsdl file defining the interface of the service.

The classes and packages under `hu.all.more.compressionervice.generated` are generated from the wsdl automatically by the source code generator tool.

#### 5.4.2. Future Developments

Further development of the current implementation is possible. Additional compression methods can be implemented. To do so, you need to modify the `compressString()`, `decompressString()` and `ListAlgorithms()` methods of the `CompressionServiceActivator` class.

## 5.5. Message Receiver Service

### 5.5.1. Component architecture/design details

This service is implemented by Applied Logic Laboratory in JAVA 1.5. This implementation is licensed as LGPL.

The source of this service can be found in the source repository as `MessageReceiverService` and `MORE_DIABALL_Message_Receiver_Client`. To run or compile this service you need to have Java version at least 1.5. To run or develop this service you need the following bundles: `eu.more.core`, `MORE_DIABALL_GMeasurement_Interface`, `eu.more.security`.

The service implements

To let this service to give you the messages it receives, you have to have a component to implement the `hu.all.more.diaball.interfaces.MessageListener` interface let the service know about this listener. The following java code is an example how to do this through the helper class `MessageReceiverCustomizer` :

```
messageReceiverCustomizer = new MessageReceiverCustomizer(context,
    messageListener);
messageReceiverTracker = new ServiceTracker(context,
    hu.all.more.diaball.interfaces.MessageProvider.class.getName(),
    messageReceiverCustomizer);
messageReceiverTracker.open();
```

### 5.5.2. Motivation for this particular design

The motivation for separating the generated classes from the actual service logic is that this way the user of the services are able to use the bundle containing the generated classes.

The `MessageListener` interface is provided to enable the cooperation between the business level (`MessageListener`) and the general service.

### 5.5.3. Future Developments

If you would like to add additional operations to this service, you need to modify the content of the wsdl file in the `MORE_DIABALL_Message_Receiver_Client` project and regenerate the generated classes there.