



FP6-IP 511731
MODELWARE
*MODELLing solution for
 softWARE systems*



WP3 Task1 - ModelBus Guidelines: Adapter

Due date of deliverable: 01/02/06

Actual submission date: 31/01/06

Start date of project: 01/08/2004

Duration: 24 months

Organisation name of lead contractor for this deliverable: LIP6

Revision: 0.8

Dissemination level:

PU	Public	<input checked="" type="checkbox"/>
PP	Restricted to other programme participants (including the Commission Services)	<input type="checkbox"/>
RE	Restricted to a group specified by the consortium (including the Commission Services)	<input type="checkbox"/>
CO	Confidential, only for members of the consortium (including the Commission Services)	<input type="checkbox"/>



**Project co-funded by the European Commission under
 the "Information Society Technologies" Sixth Framework
 Programme (2002-2006)**

Rev.	Date (dd/mm/yy)	Author	Checked by	Internal approval	Commission approval	Description
0.1	20/06/05	P. Sriplakich				Creation of document First delivered draft.
0.2	22/06/05	A. Sadovykh				Review
0.3	18/07/05	P. Sriplakich				Resolved issue about model representation as collection
0.4	19/07/05	A. Sadovykh				Revision and Release
0.5	19/10/05	A. Sadovykh				Update for the MDDI release
0.6	02/12/05	R. Pastor				Minor changes
0.7	12/12/05	A. Sadovykh				Update document for the 0.3 Version of Adapter: Custom Deploy Sessions
0.8	30/01/06	A. Sadovykh				Serialisation Tool Selection Strategies
0.9	28/07/06	A. Sadovykh N. Garande au P. Sriplak ich				Asynchronous Call Profiles

Table of contents

Table of contents	3
Figures list.....	5
1. Introduction	6
1.1. Purpose and Intended Audience	6
1.2. Structure of Document	6
2. Adapter installation.....	7
2.1. Installing standalone version	7
2.1.1. Third party libraries.	7
2.1.2. ModelBus Adapter libraries.....	7
2.2. Installing Eclipse version	7
2.2.1. Third party libraries	8
2.2.2. Adapter plugins	8
3. Adapter Validation.....	9
3.1. Registry for Adapter testing.....	9
3.2. Example Tests	9
3.2.1. Testing provider tool deployment.....	9
3.2.2. Testing Modelling Service invocation	10
3.3. Adapter Testing Framework.....	10
3.3.1. Testing provider tool deployment (activity a)	10
3.3.2. Testing Modelling Service invocation (activity b)	11
4. Tool Integration using ModelBus Adapter	12
4.1. Creating your tool description	12
4.1.1. Importing an ecore meta-model into your tool description.....	12
4.2. Implementing Modelling Services	12
4.3. Adapter Configuration and Deployment	13
4.3.1. Configuring Adapter (Consumer and Provider)	13
4.3.2. Additional steps needed for Provider tools	13
4.4. Provider role – Receiving Modeling Service invocations from other tools	14
4.5. Consumer role – Generic Way of Invoking Modeling Services.....	14

4.6.	Consumer role – Asynchronous Generic Way of Invoking Modeling Services.	15
4.7.	Default mapping between ModelBus types and Java types.....	16
4.7.1.	Flatten structure of model representation: Issue & Resolution.....	17
4.8.	Notifications.....	17
4.8.1.	Overview	17
4.8.2.	Adapter API for Notification Producers.....	19
4.8.2.1.	<i>Configure Adapter</i>	19
4.8.2.2.	<i>Create a notification</i>	19
4.8.2.3.	<i>Send the notification</i>	20
4.8.3.	Adapter API for Notification Consumers.....	20
4.8.3.1.	<i>Configure the Adapter</i>	20
4.8.3.2.	<i>Provide an object that handles received notifications</i>	20
4.8.4.	Deploy the Adapter	21
5.	Advanced Adapter Usage Topics	22
5.1.	Customising Adapter Web Service Deployment	22
5.1.1.	Customising Web Server Port	22
5.1.2.	Deploying Adapter to a Servlet Container (e. g. Tomcat).....	22
5.1.2.1.	<i>Inheriting ModelBusServlet class</i>	23
5.1.2.2.	<i>Specifying Servlet Environment</i>	23
5.1.2.3.	<i>Configuring Servlet Container</i>	24
5.2.	Sessions.....	24
5.2.1.	Declaring Provider Tool to be Session Enabled.....	24
5.2.2.	Implementing GenericSessionProvider Interface	24
5.2.3.	Using sessions on the Consumer side	25
5.2.4.	Test Case.....	25
5.3.	Customising Serialization	26
5.3.1.	Objective	26
5.3.2.	Model serialization concepts in ModelBus.....	26
5.3.3.	How ModelBus supports this approach	26
5.3.3.1.	<i>Example</i>	26
5.3.3.2.	<i>The structure of invocation message.</i>	27
5.3.3.3.	<i>ModelSerialization interface.</i>	28
5.3.3.4.	<i>How to implement a simple custom serializer ?</i>	29
5.3.3.5.	<i>How to plug your custom serializer to an adapter?</i>	29
5.3.3.6.	<i>Current Limitations</i>	29
5.3.4.	Test Case.....	29
5.4.	Tool Selection Strategies	31
5.5.	UML2 Profiles	32
5.5.1.	Background: Profile and Eclipse UML2	32
5.5.2.	How ModelBus supports UML2 profiles.	32
5.5.2.1.	<i>Objectives</i>	32
5.5.3.	API usage	33
5.5.3.1.	<i>Choosing profiles to be transmitted or not to be transmitted</i>	33
5.5.3.2.	<i>How to configure the profile resources not to be transmitted.</i>	33
5.5.3.2.1.	<i>Register profiles</i>	34

5.5.3.2.2.	Load a profiled model	34
5.5.4.	Test case	36
5.5.5.	Internal mechanism	36
5.5.5.1.	Overview	36
5.5.5.2.	Managing a set of registered profiles.....	37
5.5.5.3.	Serializing a profiled model by encoding with the global URI of the profile	37
5.5.5.4.	Deserializing a profiled model.....	37
6.	Acronyms / Dictionary	38
7.	References.....	39
8.	License Agreement	40

Figures list

Figure 1	Example scenario of using notifications	18
Figure 2	Definition of notifications	19
Figure 3	Test case	36

1. Introduction

1.1. Purpose and Intended Audience

This document describes how to install and to use ModelBus Adapter. Its intended audience is tool developers wishing to integrate their tools via ModelBus.

ModelBus Adapter implements the functionalities identified in the architecture documents [REF 2], [REF 3] and in accordance with the development plan [REF 4].

This Adapter software is located in the Eclipse MDDI workspace [REF 5], which also contains the project CVS repository.

The bug reports should be directly put to the project BugZilla system.

1.2. Structure of Document

This document contains three sections:

- Section 2 “Adapter installation” describes how to install Adapter on both plain Java and Eclipse platforms.
- Section 3 “Adapter Validation” explains how to test Adapter using its test bundle.
- Section 4 “Tool Integration using ModelBus Adapter” elaborates on how to integrate your own tools with ModelBus via its Adapter.
- Section 5 “Advanced Adapter Usage Topics” gives guidelines on advanced ModelBus features.

2. Adapter installation

2.1. Installing standalone version

In order to install the Adapter, simply include the following jar files in your CLASSPATH.

2.1.1. Third party libraries.

The third party libraries are located in subdirectory [lib](#) of the zip file. This subdirectory contains the following jar files:

- Axis library: jar files in subdirectory [lib/axis](#). These jars are distributed in Axis version 1.2 [REF 9].
- EMF library (jar files in subdirectory [lib/emf](#)). These jars are distributed in the EMF version 2.0 [REF 8]. Uml2 jars are required only when using UML2 models.
- Notification libraries (jar files in subdirectory [lib/notification](#)). This is storage for 3rd party libraries required by ModelBus Notification Service.

2.1.2. ModelBus Adapter libraries

The adapter libraries are located in subdirectory [release/stand_alone](#) of the zip file. This subdirectory contains the following jar files:

- `org.eclipse.mddi.modelbus.adapter.jar`,
- `org.eclipse.mddi.modelbus.description.jar`,

If you need to create your own Tool Description for ModelBus (4.1), we suggest you to use our Eclipse editor. Please refer to (2.2.2) for details.

For adapter validation in standalone mode, you will also need the following jars which are situated in the [test](#) folder:

- `org.eclipse.mddi.modelbus.adapter.test.jar`,
- `org.eclipse.mddi.modelbus.adapter.test.unit.jar`.

Using the libraries

The tool integration code can be developed as a Java application. It uses the API defined in `org.eclipse.mddi.modelbus.adapter.jar` in order to interact with the Adapter. It must have the `main` method for starting the application.

2.2. Installing Eclipse version

The Adapter for Eclipse platform represents a set of Eclipse plugins. In order to install the Adapter into Eclipse, copy the plugins from release bundle into your favourite Eclipse “plugins” directory. We currently support Eclipse version 3.0.

2.2.1. Third party libraries

All 3rd party libraries are distributed within the plugins. You will need EMF installed to run default ModelBus infrastructure. For UML2 model exchange the corresponding plugins should be installed.

2.2.2. Adapter plugins

The Adapter plugins are implemented by LIP6. They are located in subdirectory [release/eclipse](#) in the corresponding plugins folder. This subdirectory contains the following plugins:

- `org.eclipse.mddi.modelbus.adapter,`
- `org.eclipse.mddi.modelbus.description.`

If you need to create your own Tool Description for ModelBus (4.1), we suggest you to install ModelBus toolkit which contains the following additional plugins:

- `org.eclipse.mddi.modelbus.description.edit,`
- `org.eclipse.mddi.modelbus.description.editor.`

For Adapter validation in the Eclipse platform, please install the following plugin from [test/eclipse](#) folder too:

- `org.eclipse.mddi.modelbus.adapter.test.`

Using the plugins

The tool integration code must be contained in an Eclipse plugin. In order to use the Adapter API, this plugin will import `org.eclipse.mddi.modelbus.adapter`.

The other plugins are used internally. Therefore they do not need to be imported to the plugin of the specific-tool integration.

3. Adapter Validation

ModelBus Adapter is released with functional and unit tests bundle which is purposed to check, whether Adapter is installed properly. The bundle also demonstrates the basic functionalities and can be used as an example.

We propose a framework that enables us to build scenarios for testing the functionalities of Adapter. This framework (and an example of the use of the framework) is located as follows:

For plain Java, see

- `org.eclipse.mddi.modelbus.adapter.test.jar`,
- `org.eclipse.mddi.modelbus.adapter.test.unit.jar`.

For Eclipse version, see

- `org.eclipse.mddi.modelbus.adapter.test plugin`.

The scenarios for testing Adapter include the following activities:

- a) A tool integrator deploys the Adapter of a provider tool. The Adapter needs to be registered to the Registry in order to publish the availability.
- b) A tool integrator asks a consumer tool to consume a Modelling Service of a provider tool. The Adapter of the consumer tool needs to discover the provider via the Registry.

3.1. Registry for Adapter testing

In order to perform the presented scenario, the tool integrator must have ModelBus Registry running. The adapter is released with a test registry, which is implemented by class `org.eclipse.mddi.modelbus.adapter.test.registry.BasicRegistry`, contained in the framework. By default, the Registry can be accessed via the URL `http://your-host:8082/modelbus/modelwareRegistry`

The official ModelBus registry can be also used, however please pay attention to setup right URL.

3.2. Example Tests

We use the present framework for building a test scenario. The code implementing this scenario is located in the Java package `org.eclipse.mddi.modelbus.adapter.test.sample`.

In this scenario, a provider tool implements a Modeling Service - `"createNewClass(in className:string, in source:ModelType, out result:ModelType)"`. This service creates a class named `"className"` and add it to the input model. Then it returns the result.

The description of this provider tool is located in subdirectory [/org/eclipse/mddi/modelbus/adapter/test/sample/sample.description](#) in the test jar.

3.2.1. Testing provider tool deployment

In Java platform

Note: The registry should be run before deploying the Test Provider

The code realizing this activity is in the class `org.eclipse.mddi.modelbus.adapter.test.sample.ProviderTestSample`. It has a main method and can be executed in Java platform.

In Eclipse platform

Note: The registry should be run before deploying the Test Provider. You can start BasicRegistry from the Eclipse GUI. Click on “ModelBusTest”->”1. Registry”-> “Local Test Registry Start/Stop”. Moreover, if you use external or remote ModelBus Registry (e.g. official registry), you can specify an URL in “ModelBusTest”->”1. Registry”->”Set Remote Registry URL”

The `org.eclipse.mddi.modelbus.adapter.test` plugin implements an Eclipse GUI for deploying the Test Provider. Please use “2. Perform Provider Test” in “ModelBus Test” menu.

3.2.2. Testing Modelling Service invocation

In Java platform

The code realizing this activity is in the class `org.eclipse.mddi.modelbus.adapter.test.sample.ConsumerTestSample`. It has a main method and can be executed in Java platform.

In Eclipse platform

The Eclipse GUI for Adapter Test allows to invoke the test modelling service by clicking on “ModelBus Test”->”3. Perform Consumer Test”.

3.3. Adapter Testing Framework

In our release we provide a testing toolkit that implements the most of the preparation, deployment and invocation activities. You can reuse this toolkit for creating your own tests.

3.3.1. Testing provider tool deployment (activity a)

The class `org.eclipse.mddi.modelbus.adapter.test.ProviderTest` provides methods for testing whether the Adapter of a provider tool can be initialized and registered to the Registry.

`ProviderTest` provides the following methods.

- `setAdapter(AdapterStub adapter)` allows the tool integrator specify the Adapter he wants to test.
- `deploy()` allows the tool integrator to test whether the adapter can be deployed.
- `undeploy()` allows the tool integrator to test whether the adapter can be undeployed.

In Java platform

In order to perform the test, the tool integrator should write a main Class that does the following:

- instantiate an `AdapterStub` with appropriate properties and link it the Tool component
- invoke `setAdapter(...)`
- invoke `deploy()`
- wait for a period of time so that the consumer tools can use the Modelling Services of the provider tool
- invoke `undeploy()`

In Eclipse platform

Tool integrator creates an Eclipse plug-in that performs the same actions. These actions can be triggered by a GUI command.

3.3.2. Testing Modelling Service invocation (activity b)

The class `org.eclipse.mddi.modelbus.adapter.test.ConsumerTest` provides methods for testing whether the Adapter of a consumer tool can consume Modelling Services provided by a provider tool.

`ConsumerTest` provides the following methods.

- `setAdapter(AdapterStub adapter)` allows the tool integrator to specify the Adapter of the consumer tool.
- `setServiceName(String serviceName)` allows the tool integrator to specify the Modelling Service to be invoked.
- `setInputs(Object[] inputs)` allows the tool integrator to specify the inputs Modelling Service to be invoked.
- `consume()` allows starting the service invocation.
- `printOutputs()` allows the tool integrator to visualize the outputs of service invocation.
- `Object[] getOutputs()` allows the tool integrator to obtain the outputs of service invocation for tool-specific analysis (for example, counting model elements).

In Java platform

In order to perform the test, the tool integrator writes a main Class that do the followings:

- instantiate an `AdapterStub` with appropriate properties
- invoke `setAdapter(...)`
- invoke `setServiceName(...)`, `setInputs(...)`
- invoke `consume()`
- invoke `printOutputs()` or `getOutputs()`

In Eclipse platform

Tool integrator creates an Eclipse plugin that performs the same actions. These actions can be triggered by a GUI command.

4. Tool Integration using ModelBus Adapter

The usage of the Adapter is the same for tools that execute in Eclipse platform and in plain Java platform. The following subsections describe how to invoke Adapter API in order to deploy adapter and use it for performing the communication between tools (sending/receiving service invocations and notifications).

The following steps are required for tool integration:

1. Create a tool description;
2. Implement required Modelling Services;
3. Deploy Modelling Service with an Adapter.

The following sections describes in details these steps.

4.1. Creating your tool description

We suggest using of the Eclipse editor provided in the release package.

When the editor is installed, you can create your own tool description by following our instructions:

- Create New Project
- Create a new file description using a wizard
 - File->New->Other
 - Example EMF Model Creation Wizard->Description Model

You can import the sample description from the test bundle. For more details please refer to the architecture documentation concerning Tool Description Model [REF 3].

4.1.1. Importing an ecore meta-model into your tool description

It is possible to reuse existing EMF meta-models to define types of modelling service parameters. For this, use the import function of the Eclipse editor.

1. Choose description editor view.
2. Right-click in the editor window and choose "Load Resource"
3. In the dialog window click "Browse File System" and load external ecore meta-model you would like reference.

4.2. Implementing Modelling Services

In order to integrate tools to ModelBus, tool integrators must provide the implementation of the Tool component (see [REF 3]). Concretely, they must provide classes that implement the following Java interfaces:

`org.eclipse.mddi.modelbus.adapter.user.provider.GenericProvider`: for tools that provide sessionless Modelling Services.

`org.eclipse.mddi.modelbus.adapter.user.provider.GenericSessionProvider`: for tools that provide sessionful Modelling Services.

`org.eclipse.mddi.modelbus.adapter.user.notification.NotificationConsumer`: for tools that expect the reception of notifications.

In order to play the role of consumer tools, or to emit notifications, the tools do not need to implement any interfaces.

4.3. Adapter Configuration and Deployment

The Adapter deployment allows a tool to be plugged to ModelBus. The adapter configuration is required for the both Provider and Consumer tools, while adapter deployment is mostly required on Provider side. If the tool is a Notification Consumer, the adapter should also be deployed.

4.3.1. Configuring Adapter (Consumer and Provider)

Instantiate the Adapter.

In order to instantiate the Adapter, the following properties must be defined:

- "registry_location" URL specifying the Web Services location of the Registry.
- "notification_service_location" URL specifying the Web Services location of the NotificationService
- "tool_desc_file" Path location of tool description file in XMI format. The tool description is an instance of the ModelBus metamodel.

The key of these properties are defined as constants in `AdapterStubImpl` class (`PROP_REGISTRY_LOCATION`, `PROP_NOTIF_LOCATION`, `PROP_TOOL_DESC_FILE`).

```
import org.eclipse.mddi.modelbus.adapter.user.AdapterStub;

Properties p = new Properties();
p.put(AdapterStub.PROP_REGISTRY_LOCATION, "http://some-
host:8082/modelbus/modelwareRegistry");
p.put(AdapterStub.PROP_NOTIF_LOCATION, "http://server2/NotificationService");
p.put(AdapterStub.PROP_TOOL_DESC_FILE, "../resources/tooldescription.xml");

AdapterStub adapter = new AdapterStubImpl(p);
```

4.3.2. Additional steps needed for Provider tools

The following steps are required only for tools that provide Modelling Services and/or expect the reception of notifications. If it is not the case, then these steps can be skipped.

- 1) Instantiate classes that implemented Tool interfaces (described in **Erreur ! Source du renvoi introuvable.**).

The instantiation mechanism is tool-dependent. We suppose that the created instances are assigned to the following variables.

```
import org.eclipse.mddi.modelbus.adapter.user.notification.NotificationConsumer;
import org.eclipse.mddi.modelbus.adapter.user.provider.GenericProvider;
import org.eclipse.mddi.modelbus.adapter.user.provider.GenericSessionProvider;

GenericProvider provider = ...
GenericSessionProvider sessionProvider = ...
NotificationConsumer nc = ...
```

2) Link these instances to the Adapter.

```
adapter.getToolStub().setProvider(provider);
adapter.getToolStub().setSessionProvider (sessionProvider);
adapter.getToolStub().setNotificationConsumer(nc);
```

3) Make the Adapter available on the Web Services server.

Note: The address of web service is set automatically by the Adapter. The default url is:

http://YOUR_HOST:8081/modelbus/modeling_services/YOUR_TOOL_NAME

You can change default port or deploy Adapter to a servlet container. These topics are covered by 5.1.

```
import org.eclipse.mddi.modelbus.adapter.user.AdapterStub;

adapter.deploy();
```

4.4. Provider role – Receiving Modeling Service invocations from other tools

When a tool invokes a Modelling Service, the instance of `GenericProvider` or `GenericSessionProvider` interfaces (referred to as “provider”, “sessionProvider” in the previous example) will be executed.

4.5. Consumer role – Generic Way of Invoking Modeling Services.

In order to invoke Modelling Services, the tool integrator must write code that invokes Adapter API as follows.

```
Object[] inputs = ...; // ...
try {
    Object[] outputs = adapter.getGenericConsumer()
                              .consume("checkOCL", inputs);
} catch (ServiceUnknownException e) {
    e.printStackTrace();
} catch (NoToolAvailableException e) {
    e.printStackTrace();
} catch (ModelTypeMismatchException e) {
    e.printStackTrace();
} catch (ModelBusCommunicationException e) {
    e.printStackTrace();
}
```

The variables “inputs” and “outputs” is the list of parameter values of the Modelling Service.

- “inputs” means the parameters with the directions “in” and “inout”.
- “outputs” means the parameters with the directions “return”, “out” and “inout”.
- The order of the values in the array is determined by the order of parameter declared in the Modelling Service description.

Please note that the first parameter of the consume method, `serviceName`, can be a qualified name. This means that it is possible to fully specify the service by the Tool name, Interface name and Modelling Service name. In this case modelling service name is delimited by “.”. For example, for OCLCheck service the qualified name can be “OSLO.interface1. checkOCL”.

4.6. Consumer role – Asynchronous Generic Way of Invoking Modeling Services.

In order to invoke asynchronously Modelling Services, the tool integrator must write code that invokes Adapter API as follows.

```
Object[] inputs = ...; // ...
try {
    GenericConsumer consumer = adapter.getGenericConsumer();
    String conId = consumer.consumeAsync("checkOCL",inputs);

    while(!consumer.isResultReady(conId){
        // result not ready, then do something else ...
    }

    // result ready
    Object[] outputs = consumer.getResult(conId);
} catch (ServiceUnknownException e) {
    e.printStackTrace();
} catch (NoToolAvailableException e) {
    e.printStackTrace();
} catch (ModelTypeMismatchException e) {
    e.printStackTrace();
} catch (ModelBusCommunicationException e) {
    e.printStackTrace();
} catch (ModelingServiceError e) {
    e.printStackTrace();
}
```

The variables “inputs” and “outputs” are the same used in the generic synchronous way invocation. Please note that a synchronous consummation is directly linked with a connection identifier, then it is possible for one consumer to invoke several asynchronous modeling services simultaneously.

Test Case

In order to illustrate *asynchronous calls* usage, we provide examples in the `org.eclipse.mddi.modelbus.adapter.test.asynchronous` package.

`SortModelingServiceImpl` provides the modeling service “sort” which increases the first parameter before returning it.

`AsyncProvider` is a provider of the `SortModelingService`.

`AsyncConsumer` is a consumer that consumes sort modeling service in an asynchronous way. The consumer launches the asynchronous connection and tries several times to know if the result is ready. When the result is ready, the consumer gets the result.

4.7. Default mapping between ModelBus types and Java types

This section describes the default mapping between ModelBus and Java types. It is possible to override this mapping using custom serializers mechanism described in 5.3.

A parameter can be either single-valued or multi-valued. It is single-valued if its `upper` property is one. Otherwise, it is multi-valued (`upper=-1` or `upper>1`).

We use the following mappings for single-valued parameters and multi-valued parameters:

Single-valued parameters

ModelBus types	Java Types
PrimitiveType String, Boolean, Integer, Double, Binary	java.lang.String, java.lang.Boolean, java.lang.Integer, java.lang.Double, java.io.InputStream
EnumerationType	java.lang.String The string represents the literal value of the enumeration.
ModelType	java.lang.Collection The collection contains instances of org.eclipse.emf.ecore.EObject (see EMF [REF 6]). <u>The collection contains all model elements (EObject) in the model, including the sub model elements.</u> For example, a UML model contains packages and classes inside the packages. The collection representing this model must contain all packages and all classes in this model. (See 4.7.1 for more details) In the case of “inout” parameters, the members of the collections can be modified when passed to the Modelling Service.

Multi-value parameters

A Java array is used for representing the value of multi-valued parameters. Elements in the Java array represent the elements of the parameter value.

ModelBus types	Java Types
PrimitiveType name="string" name="boolean" name="integer" name="double"	The Java type is determined by the name of the <i>PrimitiveType</i> instance. java.lang.String[], java.lang.Boolean[], java.lang.Integer[], java.lang.Double[]

<i>EnumerationType</i>	<code>java.lang.String[]</code>
<i>ModelType</i>	<code>java.lang.Collection[]</code>

Null value representation

Single-valued parameters ([upper..lower] = [0..1] or [1..1])

- In the case of [0..1], null parameter value is represented by `null` Java value.
- In the case of [1..1], the parameter value can not be null

Multi-valued parameters

- The null parameter value is represented by an array of size zero.

4.7.1. Flatten structure of model representation: Issue & Resolution

In ModelBus, we have chosen to represent models as a collection containing models elements in a flatten structure. This means that the collection must contains all model elements including sub model elements. The reason for this is to support fine-grained model representation. It is possible to represent parts of models (excluding some sub elements of the original models).

However, some tool integrators (who do not to wish represent partial models) may find this approach impractical for the following reasons:

- 1) Tools need to flatten the structure of models before passing models to Adapter.
- 2) Tools need to rebuild the hierarchical structures of models when received models from Adapter.

In order to facilitate the tool integration, we have provided an API for flattening / rebuild hierarchical structures of models. This API includes two operations in the class `org.eclipse.mddi.modelbus.adapter.impl.model_manipulation.ModelUtil`:

```
1. public static Collection flattenHierarchicalCollection (Collection input)
```

This operation is used for flattening hierarchical structure of models. The returned collection includes all model elements in the input collection and their sub model elements.

```
2. public static Collection getTopElements (Collection input)
```

This operation is used for rebuilding hierarchical structure of models. The returned collection excludes all sub model elements. In other words, it contains only top-most model elements of the models elements contained in the input collection.

By using both operations, tool integrators can easily switch between two styles of model representations (flatten or hierarchical).

4.8. Notifications

4.8.1. Overview

A notification is a piece of information that a tool wants to broadcast to a group of tools that are interested in this kind of information. The distribution of the notifications is managed by Notification Broker.

Tools do not communicate directly with the Notification Broker, but via their Adapter. The added-value is that the Adapter hides the technical difficulties (such as Web Services communication) and also provides model-dedicated facilities (notifications can contain models).

Figure 1 shows an example scenario of using notifications. Objecteering has the role of **Notification Producer**. It publishes a notification containing the information about changes in models. The Adapter of Objecteering will then send this notification to NotificationService. NotificationService maintains the set of **Notification Consumers** (i.e. tools that are interested in particular kind of notifications). In this scenario, OCLTool is interested in notifications about model changes. Therefore, once the notification is sent from Objecteering, it is delivered to OCLTool.

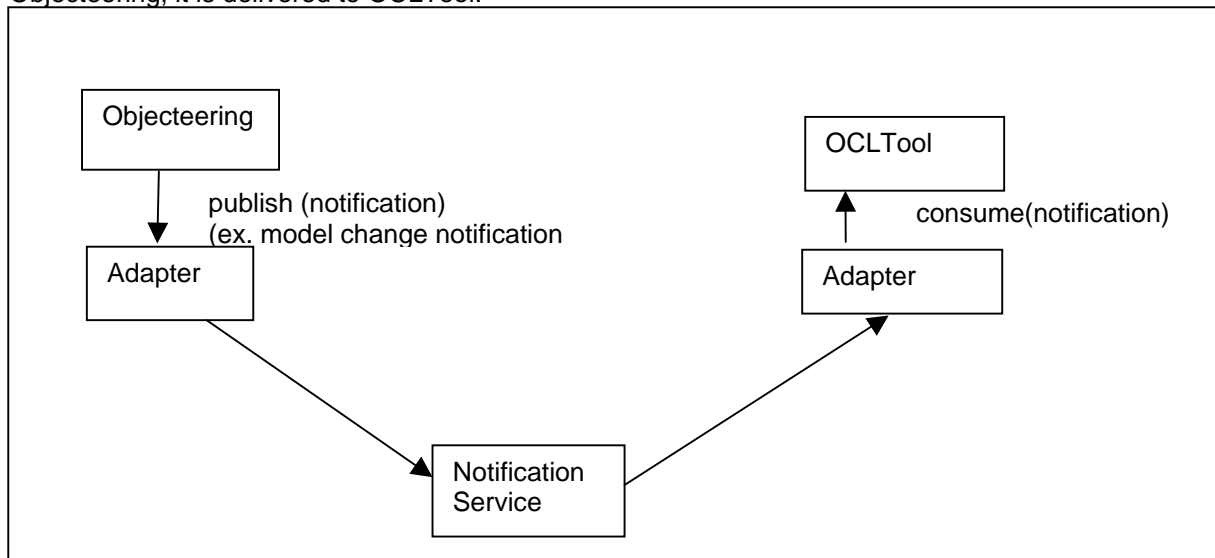


Figure 1 Example scenario of using notifications

The concept of notifications is defined in ModelBus Software Architecture Documentation [REF 3]. The information contained in notifications is defined in *EventType* metaclass in ModelBus metamodel (figure below). The name of EventType is used to identify the kind of information (for ex. “ModelChangeNotification”, “ModelCreatedNotification” etc). EventType is linked with Type, so we can define the information contained in notifications. For example, an EventType can be linked with ModelType, meaning that corresponding notifications can contain models.

In order to link notifications and tools, Tool integrators should describe the notifications produced by the tools by linking ModelingServiceInterface with EventType. On the other hand, information about the notification consumers (e.g. OCLTool consumes model change notification) is not expressed in this metamodel. This allows the notification consumers to express their interest in notifications in a dynamic way.

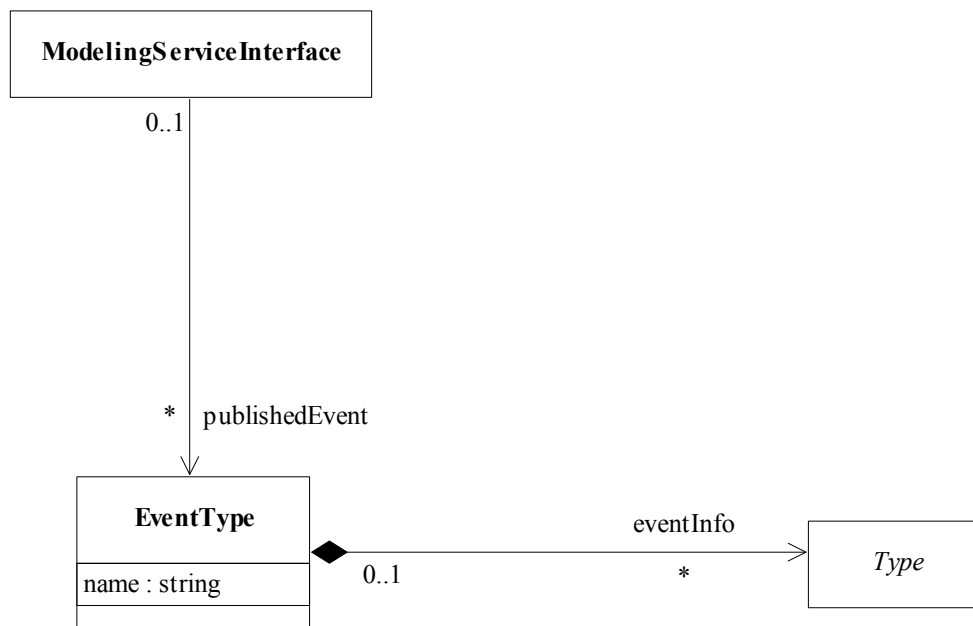


Figure 2 Definition of notifications

4.8.2. Adapter API for Notification Producers

4.8.2.1. Configure Adapter

The tool integrator needs to configure the Adapter so that it knows the address of Notification Broker. This configuration is done at the instantiation of AdapterStub (see 4.3.2).

```

Properties p = new Properties();
p.put(AdapterStub.PROP_NOTIF_LOCATION, "http://localhost:8080/Notification");

AdapterStub adapter = new AdapterStubImpl(p);
  
```

4.8.2.2. Create a notification

Notifications are concretely represented as instances of the class `org.eclipse.mddi.modelbus.adapter.user.notification.Notification` and have two properties: `name` and `info`. The property `name` identifies the kind of notifications (according to `EventType`). The property `info` stores the information contained by the notification, which must conform to the type defined in `EventType`. This information can be string, boolean, model, etc. The Java representation of the information is defined in 4.7.

The following code illustrates how to create a notification.

```

Notification notification = new Notification("ModelChangedNotification",
"Modell");
// This notification informs that Modell is changed.
  
```

4.8.2.3. *Send the notification*

The following codes illustrate how to send the notification using the adapter.

```
// 'adapter' is instance of AdapterStub
NotificationPublisher pub = adapter.getNotificationPublisher();
pub.publish(notification);
```

4.8.3. **Adapter API for Notification Consumers**

4.8.3.1. *Configure the Adapter*

Similarly to Notification Producers the Notification Consumers need to configure the location of the Notification Broker. This configuration is done in a similar way (using Properties of the Adapter).

Moreover, Notification Consumers need to subscribe to particular kinds of notifications. In the metamodel in Figure 2, we have shown that different kinds of notifications can be identified with the concept of *EventType*. More precisely, an *EventType* corresponds to a kind of notifications.

Concretely, we use the comma-list string to express the kinds of notifications in which a Notification Consumer is interested. For example, “ModelChangedNotification, ModelCreatedNotification”, where “ModelChangedNotification” and “ModelCreatedNotification” are the names of two *EventTypes*. This comma-list string is put in the property of the Adapter, as illustrated in the following code.

```
// p is instance of Properties
p.put(AdapterStub.PROP_NOTIF_LOCATION, "http://localhost:8080/Notification");
p.put(AdapterStub.PROP_NOTIF_TOPICS,
"ModelChangeNotification,ModelCreatedNotification");

AdapterStub adapter = new AdapterStubImpl(p);
```

4.8.3.2. *Provide an object that handles received notifications*

Notification Consumer is a tool that wants to receive a particular kind of notifications. Upon the reception of notifications, the adapter invokes an object that is provided by the Notification Consumer. Then, this object can perform tool-specific process of the received notifications.

The tool integrator needs to provide an object implementing the interface `org.eclipse.mddi.modelbus.adapter.user.notification.NotificationConsumer`. This interface has the following signature:

```
public interface NotificationConsumer {
    public void consume(Notification notif);
}
```

The operation `consume` is invoked when a notification is received.

The object can be linked to the adapter as follows

```
// suppose that NotificationConsumerImpl implements NotificationConsumer
// It is provided by the tool integrator
NotificationConsumer con = new NotificationConsumerImpl();
// link to the Adapter
adapter.getToolStub().setNotificationConsumer(con);
```

4.8.4. Deploy the Adapter

Tool integrator needs to deploy the adapter so that it begins listening to notifications emitted from Notification Broker. This step is similar as the deployment of the tools providing Modelling Services. It is illustrated as follows:

```
adapter.deploy();
```

5. Advanced Adapter Usage Topics

This section discusses several advanced topics.

5.1. Customising Adapter Web Service Deployment

In many cases the ModelBus deployment should be customised. For example, in case several ModelBus Adapter instances are required or Adapter is to be installed to a servlet container.

This section describes how to customise deployment.

5.1.1. Customising Web Server Port

By default, ModelBus Adapter listens port 8081 for incoming messages (service invocations, notifications). In some cases it can be incontinent, for example when this port is already occupied – instance of ModelBus adapter is already run, another allocation use port 8081, etc.

The Adapter API permits to set-up the port for deployment. Use “modelbus_port” (defined in AdapterStub.MODELBUS_PORT constant) property to specify a new port before deploying adapter stub.

```
Properties p = new Properties();  
  
...  
String newPort = "5555";  
  
p.put(AdapterStub.MODELBUS_PORT, newPort);  
  
AdapterStub adapter = new AdapterStubImpl(p);  
  
...  
  
adapter.deploy();
```

Please see `org.eclipse.mddi.modelbus.adapter.deploy.CustomDeployTool` class for an example.

5.1.2. Deploying Adapter to a Servlet Container (e. g. Tomcat)

ModelBus adapter can be deployed to a standard servlet container to reuse existing Web Server infrastructure or ensure performance. This section describes how to deploy Adapter as a servlet to the Tomcat server.

For integrating ModelBus to an existing servlet environment, we suggest the following approach:

1. Inherit `ModelBusServlet`;
2. Override `init` method in order to specify your servlet environment and to deploy all your modelling services;
3. Configure servlet container to redirect all messages coming to `/modeling_services/...` and `/notification/...` to your new servlet.

Please see `org.eclipse.mddi.modelbus.adapter.test.tomcat` for example.

5.1.2.1. Inheriting ModelBusServlet class

org.eclipse.mddi.modelbus.adapter.infrastructure.transport.ws.server.ModelBusServlet is an abstract class that implements dispatching of incoming SOAP requests. This class should be inherited in order to specify servlet environment and to deploy your modelling services. SampleToolDeployServlet from tomcat example is defined as follows:

```
public class SampleToolDeployServlet extends ModelBusServlet
```

5.1.2.2. Specifying Servlet Environment

By default, ModelBus adapter uses “modelbus” context and “8081” port. If you like to deploy adapter to different context and to a server that uses different port, you should specify these parameters. ModelBusServlet class has setDefaultPort and setDefaultContext that override default settings. You can use them when initialising servlet. For this override init method.

After this the servlet should be registered as a ModelBus server in the Adapter Container. For this, use ModelBusServlet method registerServer.

This all is illustrated in the SampleToolDeployServlet example:

```
public void init(ServletConfig config) {  
    setDefaultPort(8080);  
    setDefaultContext("modelbus_tomcat");  
    registerServer();  
    ...  
}
```

Thus, the servlet environment is specified. The modelling deployment is quite the same as for other cases 4.3.2. In the case of SampleToolDeployServlet the init method contains the following:

```
    ...  
    String registry_loc = ...;  
    // put description file here  
    String desc_file = ...;  
    //load properties  
    Properties p = new Properties();  
    p.put(AdapterStub.PROP_REGISTRY_LOCATION, registry_loc);  
    p.put(AdapterStub.PROP_TOOL_DESC_FILE, desc_file);  
  
    //deploy tool as usual  
    AdapterStub adapter;  
    try {  
        adapter = new AdapterStubImpl(p);  
        GenericProvider provider = new SampleModelingServiceImpl();  
        adapter.getToolStub().setProvider(provider);  
        adapter.deploy();  
    } catch (DeploymentException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
}
```

5.1.2.3. Configuring Servlet Container

In the case of Tomcat, each context (i.e. web application) should contain web.xml file, declaring all servlets. This file is also used to specify redirection of incoming calls. Please see the following example.

```
<servlet-mapping>
  <servlet-name>SampleToolDeployServlet</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

In this example all incoming calls for URL with pattern “/*”, i.e. all calls are redirected to our servlet, that in turn dispatch it to the Adapter.

Actually, only “/modeling_services/*” and “/notification/*” patterns should be redirected. Therefore, in case the context is used for other servlets too, the following mapping can be applied.

```
<servlet-mapping>
  <servlet-name>SampleToolDeployServlet</servlet-name>
  <url-pattern>/modeling_services/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>SampleToolDeployServlet</servlet-name>
  <url-pattern>/notification/*</url-pattern>
</servlet-mapping>
```

5.2. Sessions

ModelBus Adapter implements a session mechanism as specified in [REF 3].

org.eclipse.mddi.modelbus.adapter.test.session package contains a sample tool pair that illustrate session usage. The following sections describe session mechanism using this example.

5.2.1. Declaring Provider Tool to be Session Enabled

In order to declare a provider tool to be “session enabled”, it is required to add property “IsSessionEnabled”, which should be equal to “true”. Please see org.eclipse.mddi.modelbus.adapter.test.session.session_tool.description for example:

```
<content xsi:type="concrete:Tool" name="SampleSessionTool"
interface="//@content.0">
  <property name="IsSessionEnabled" value="true"/>
</content>
```

5.2.2. Implementing GenericSessionProvider Interface

org.eclipse.mddi.modelbus.adapter.user.provider.GenericSessionProvider interface should be implemented in order to connect a tool as a session enabled provider.

This interface contains three method signatures that should be implemented by the Tool Integrator:

- `String newSession()` – is called each time a consumer asks for a new session. Provider tool should return a session id;
- `void endSession(String sessionId)` – is called each time a consumer asks to end a session specified by an id;
- `Object[] executeSession (String serviceName, Object[] parameterValues, String sessionId)` – is a entry point for all incoming service invocations. Session id is specified for each call.

Please see `org.eclipse.mddi.modelbus.adapter.test.session.SampleSessionTool` for example.

5.2.3. Using sessions on the Consumer side

Configure adapter as usual (4.3.1).

If a full control of sessions is not necessary, a consumer tool can call provider tool in the exactly the same way as described in 4.5.

In order to fully control sessions, please use the following session dedicated methods of `org.eclipse.mddi.modelbus.adapter.user.consumer.GenericConsumer`:

- `String newSession()` – will return an id of a new session;
- `endSession(String sessionId)` – will end the session;
- `String[] getOpenSessions(String serviceName)` – will return a set of open sessions for a tool specified by `serviceName`. **Note** that `serviceName` can be qualified as described in 4.5.
- `invoke (String serviceName, Object[] parameterValues, String sessionId)` – will execute a modelling service within a session specified by the `sessionId`.

Please see `org.eclipse.mddi.modelbus.adapter.test.session.SampleSessionConsumer` for example:

```
consumer.consume("SampleSessionTool.interface1.openPackage", new Object[] {
"Pack1" }, s1);
```

5.2.4. Test Case

In order to illustrate *session* usage, we provide an example in the `org.eclipse.mddi.modelbus.adapter.test.session` package.

In this test case the following scenario is implemented.

`SampleSessionTool` provides 2 modelling services:

- `openPackage` – creates/opens a package in a given model. During this operation the package name is associated with the session;
- `addNewClass` – adds a class to the open packages, which is determined by session id.

`SampleSessionConsumer` opens new sessions, opens packages and ask to add a new class to the model. `SampleSessionTool` adds classes to open packages depending on session in which the `addNewClass` is called.

In order to illustrate session usage in an asynchronous way, we provide the `org.eclipse.mddi.modelbus.adapter.test.asynchronous.AsyncSessionProvider` and `AsyncSessionConsumer` classes. The scenario is the same described in the session Test Case part.

5.3. Customising Serialization

5.3.1. Objective

The objective of this document is to define the model serialization interface of ModelBus.

Tool integration can implement this interface to perform a custom serialization (i.e. conversion between XML and other object representation).

5.3.2. Model serialization concepts in ModelBus

In ModelBus, a modeling service parameter can be “a set of references to any elements in any models”. This approach enables us to specify several parameters pointing to elements in the same model. For example, we can define three parameters pointing to three classes (C1, C2, C3) in the same class diagram (this class diagram can contain other classes as well).

Consequently, in an invocation message, a parameter value is not equivalent to an XML document. For example, the following cases are possible

- A parameter value can refer to any elements of a XML document, not only top elements.
- Several parameter values can point the elements of the same XML document.
- A single parameter value can contains references to elements from several XML documents

This approach gives more freedom to the modeling service definition.

5.3.3. How ModelBus supports this approach

In this approach, there is no mapping 1-to-1 from parameter values to XML documents. To encode service invocation message, ModelBus serializes all XML documents first. Then, ModelBus write the parameter values as the references to the elements in the XML documents previously serialized.

5.3.3.1. Example

This is an example invocation message. This message contains a class diagram with 5 classes (class1, class2, class3, class4, class5). It contains two parameters (parameter1 and parameter2). The value of parameter1 is the references to class1 and class2. The value of parameter2 is the reference to class3.

ModelBus serializes the class diagram in the XML tag “scope”. Then, ModelBus serializes the parameter values as the references to the model elements in the scope.

```
<modelbus:service1 xmlns:modelbus="http://www.eclipse.org/mddi/modelbus">
  <modelbus:Scope>
    <modelbus:Resource uri="default_resource.modelbus">&lt;?xml version=&quot;1.0&quot;
encoding=&quot;ASCII&quot;?&gt;
&lt;xmi:XMI xmi:version=&quot;2.0&quot; xmlns:xmi=&quot;http://www.omg.org/XMI&quot;
xmlns:ecore=&quot;http://www.eclipse.org/emf/2002/Ecore&quot;&gt;
  &lt;ecore:EClass name=&quot;class1&quot;/&gt;
  &lt;ecore:EClass name=&quot;class2&quot;/&gt;
  &lt;ecore:EClass name=&quot;class3&quot;/&gt;
  &lt;ecore:EClass name=&quot;class4&quot;/&gt;
  &lt;ecore:EClass name=&quot;class5&quot;/&gt;
&lt;/xmi:XMI&gt;
    </modelbus:Resource>
  </modelbus:Scope>
  <modelbus:parameter1 encoding="modelbus:ModelEncoding">
```

```

    <modelbus:Ref uri="default_resource.modelbus" fragment="/0"/>
    <modelbus:Ref uri="default_resource.modelbus" fragment="/1"/>
  </modelbus:model>
  <modelbus:parameter2 encoding="modelbus:ModelEncoding">
    <modelbus:Ref uri="default_resource.modelbus" fragment="/2"/>
  </modelbus:model>
</modelbus:createNewClass>
</soapenv:Body>

```

5.3.3.2. The structure of invocation message.

The invocation message contains two parts.

- The first part “scope” contains the contents of all XML documents to be transmitted in this message. Each XML document is identified with a URI.
- The second part contains all parameter values. For ModelType, a parameter value contains the references to the elements in the XML documents in the first part. A model element is referenced with the tuple (uri, fragment). The uri identifies the XML document containing this element. The fragment identifies this element inside the XML document.

Invocation message (request and reply)

<modelbus:Scope>: contains all XML documents to be transmitted

<modelbus:Resource> : contains an XML document.

attribute “**uri**”: the identifier of this XML document

text value: the content of the XML document

<modelbus:parameter1>: contains the parameter value. For ModelType, a parameter value is a set references to model elements in the scope.

<modelbus:Ref>: represents the reference to one model element.

attribute “**uri**”: the uri of the XML document that contains the referenced model element.

attribute “**fragment**”: the identifier of the model element inside the specified XML document. This identifier can be path to the element (such as “/0”) or the unique ID of the element.

<modelbus:Ref> : another reference

.....

<modelbus:parameter_2>

.....

<modelbus:parameter_n>

5.3.3.3. *ModelSerialization interface.*

The following interface enables ModelBus to construct the invocation message. ModelBus provides a default implementation of this interface for serializing models represented as EMF objects. However, the tool integrator can implement this interface to support the serialization of models in other formats than EMF.

```
public interface ModelSerializer {
    public SerializedXmiDocument[] serialize(Parameter[] params, Object[] values);
    public DeserializedModel[] deserialize(SerializedXmiDocument[] documents);
    public ModelElementReference[] getReferences(SerializedXmiDocument[] documents,
        Parameter p, Object o);
    public Object dereference(DeserializedModel[] data, Parameter p, ModelElementReference[] refs);
}
```

```
public class
SerializedXmiDocument {
    String uri;
    //the uri of the document
    String xmi;
    //the serialized content
    //have methods get/set
}
```

```
public class
DeserializedModel {
    String uri; //the uri of the
document from which model
is serialized.
    Object value; //the
deserialized value of the
model
    //have methods get/set
}
```

```
public class
ModelElementReference {
    String uri; //the uri
identifying the document
    String ref; //the fragment
identifying the model
element
    //have methods get/set
}
```

Interface ModelSerializer contains the following methods:

- **public SerializedXmiDocument[] serialize(Parameter[] params, Object[] values);**
serialize all XMI documents referred to by all parameter values. Note that several parameters values may refer to common XMI documents, or a parameter value may refer to several XMI documents. The serialized documents are presented with the class SerializedXmiDocument. This class contains the URI of the document and the serialized value
- **public DeserializedModel[] deserialize(SerializedXmiDocument[] documents);**
deserialize the XMI documents to produce the object representation. This is the inverse process of "serialize".
- **public ModelElementReference[] getReferences(SerializedXmiDocument[] documents, Parameter p, Object o);**
obtain a set of references corresponding to a parameter value. A reference consists of a URI identifying the XMI document and the fragment identifying the element in the document. The reference is represented with the class ModelElementReference.
- **public Object dereference(DeserializedModel[] data, Parameter p, ModelElementReference[] refs);**
obtain model elements corresponding to the references. This is the inverse process of "getReferences".

5.3.3.4. *How to implement a simple custom serializer ?*

This ModelSerializer interface is powerful but complex. We also suggest a simple way to implement it. In this case, the custom serializer can extend the abstract class **AbstractBasicModelSerializer**. This class implements a part of the ModelSerializer interface. If the tool integrator bases his implementation of this class, then, he only needs to implement two abstract methods.

- **abstract protected void serialize(Object o, OutputStream xmiStream);**
serialize an object (parameter value) to an XMI document.
- **abstract protected Object deserialize(InputStream xmiStream);**
deserialize the XMI document to an object form.

Those two methods are compatible the Serializer interface of ModelBus.

AbstractBasicModelSerializer offers more simplicity for implementing a custom serializer. However, it has the following restrictions:

- Mapping between parameters and XMI document is 1-to-1. I.e., a parameter value corresponds to an XMI document.
- All parameter values (of ModelType) refer to the top model elements of the XMI documents.
- Each XMI documents do not have links with each other : They are independent.

5.3.3.5. *How to plug your custom serializer to an adapter?*

You should call the adapter method **setModelSerializer**.

Note, that **setSerializer** is now deprecated.

For example for a given custom serializer which is implemented by a CustomSerializer class, the initialisation routine can look like:

```
adapter = new AdapterStubImpl(p);  
//setting up new serializer  
adapter.setModelSerializer(new CustomSerializer());
```

For more information, please refer to the **adapter.test** project into the **org.eclipse.mddi.adapter.test.serializer** package.

5.3.3.6. *Current Limitations*

Using this approach, all serialized information is correctly deserialized on a remote side. However, as it was noticed during the tests, when on the remote side model is a collection of elements belonging to several EMF resources, the ModelBus default serializer puts them to several **scope** sections. Such a document can't be correctly parsed by **AbstractBasicModelSerializer**. Please use **ModelSerializer** or wait for a better implementation.

5.3.4. **Test Case**

The source code of the example can be found in `org.mddi.modelbus.adapter.test.serializer`.

In this example, the new serializer permits to keep model in a String format. The serializer is applied on the consumer side, while provider side still represents models in EMF format.

In order to illustrate interoperability, the ProviderTestSample (section 3) can be used on the provider side.

The CustomSerializer is implemented as follows:

```
...
protected void serialize(Object o, OutputStream xmiStream) {
    String myXMI = (String) o;

    try {
        xmiStream.write(myXMI.getBytes());
    } catch (IOException e) {
        e.printStackTrace();
    }
}

/*
 * (non-Javadoc)
 *
 * @see
 * org.eclipse.mddi.modelbus.adapter.infrastructure.serialize.AbstractBasicModelSer
 * ializer#deserialize(java.io.InputStream)
 */
protected Object deserialize(InputStream xmiStream) {

    try {
        String myXMI;
        int n = xmiStream.available();
        byte buffer[] = new byte[n];
        xmiStream.read(buffer);
        myXMI = new String(buffer);
        return myXMI;
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    return null;
}
...
```

Then the consumer or provider can work with models directly in String format.

The following consumer implementation intends to illustrate this mechanism:

```
...
String model = "<?xml version=\"1.0\" encoding=\"ASCII\"?>"+
    "<xmi:XMI xmi:version=\"2.0\"
xmlns:xmi=\"http://www.omg.org/XMI\"/>"; // model in String format

...
    Properties p = new Properties();
    p.put(AdapterStub.PROP_REGISTRY_LOCATION, registry_loc);

    adapter = new AdapterStubImpl(p);
    //setting up new serializer
    adapter.setSerializer(new CustomSerializer());

    consumer = adapter.getGenericConsumer();
```

```

        for (int count = 0; count < 4; count++) {
            String newClassName = "class A" + count;
            logger.info("Create New Class with name:"+newClassName);
            Object[] outputs = consumer.consume(createNewClass, new Object[]
{ newClassName,model });
            model = (String) outputs[0];
            logger.info("Result Model:" + model);
        }
    }
}

```

The `model` is processed as an XMI string on the consumer side, while on the provider side the `ProviderTestSample` process the same `model` as EMF Object.

The example should be executed in the following order:

1. Run `org.mddi.modelbus.adapter.test.registry.BasicRegistry`
2. Run `org.mddi.modelbus.adapter.test.sample.ProviderTestSample`
3. Run `org.mddi.modelbus.adapter.test.serializer.SerializerConsumer`

5.4. Tool Selection Strategies

Every service invocation operation starts with tool selection procedure, whereas a presumably right tool is selected from a pool of registered tools. Providing that several registered tools provide equivalent services, different selection strategies can be applied. The default strategy is to look for the local tools available and then the other tools. The first found tool is returned as selected one.

Taking into account that this strategy may be not optimal for some cases, we provide a mechanism to customise this strategy.

The user should implement the `org.eclipse.mddi.modelbus.adapter.infrastructure.registry.ToolSelectionStrategy` interface that is defined as follows:

```

public interface ToolSelectionStrategy {

    public Tool selectTool(String serviceName, Collection tools);

}

```

This interface defines `selectTool` method which is called for the tool choosing. It should return a `Tool` object, which is specified in the ModelBus EMF meta-model. `serviceName` is a located service, while `tools` are a Collection of Tool objects retrieved from the registry.

Note: Please note that `serviceName` can be specified as a qualified service name, c.f. Acronyms / Dictionary in section 6.

Once Tool Selection Strategy is implemented, it can be set up in the adapter in any moment, by calling adapter's method `setToolSelector`. For example, please consider the following code:

```

...
        Adapter adapter = new AdapterStubImpl(p);
...
        //setting up new tool selection strategy
        adapter.setToolSelector(new NewToolSelectionStrategy());
...

```

5.5. UML2 Profiles

5.5.1. Background: Profile and Eclipse UML2

Eclipse UML2 is an EMF-based tool that provides an API and a GUI for manipulating UML models. It enables a user to work with UML profiles in the following ways:

Profile definition: User can define his own profile with the provided GUI and save it as an XMI file. By convention the profile is saved with the extension *.profile.uml2.

Profiled model creation: The provided GUI is used to create a UML model based on the previously created profile. A UML model that is based on a profile is also referred as to a “*profiled model*”. A profiled model is similar to the basic UML model but it contains extra information. The extra information is represented in the form of “*tag values*” attached to each UML model element. By convention, a profiled model is saved with the extension *.uml2.

A UML model references to the profile upon which it is based. In the XMI file, this reference is represented as a URI link to the profile XMI file (.profile.uml2).

The following XMI code gives an example of a profiled model. The URI referencing the profile XMI file is bold. The extra information representing tag value is in blue.

```
<uml:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:network_0="http://network_0.profile.uml2"
xmlns:uml="http://www.eclipse.org/uml2/1.0.0/UML"
xsi:schemaLocation="http://network_0.profile.uml2
src/profile/MyProfile.profile.uml2#_zjVyIEYuEdqKKKoghK_EBg"
xmi:id="_JM0G4EYtEdqKKKoghK_EBg" name="mymodel" appliedProfile="_3p-
3YEUyEdqKKKoghK_EBg">
  <packageImport xmi:type="uml:ProfileApplication" xmi:id="_3p-
3YEUyEdqKKKoghK_EBg">
    <eAnnotations xmi:id="_3qE-AEYuEdqKKKoghK_EBg" source="attributes">
      <details xmi:id="_3qE-AUYuEdqKKKoghK_EBg" key="version" value="0"/>
    </eAnnotations>
    <importedPackage xmi:type="uml:Profile"
href="MyProfile.profile.uml2#_9PAf0EYmEdqKKKoghK_EBg"/>
    <importedProfile href="MyProfile.profile.uml2#_9PAf0EYmEdqKKKoghK_EBg"/>
  </packageImport>
  <ownedMember xmi:type="uml:Package" xmi:id="_P9ESsEYtEdqKKKoghK_EBg"
name="company1">
    <ownedMember xmi:type="uml:Class" xmi:id="_R140cEYtEdqKKKoghK_EBg"
name="ProductInfoServer">
      <eAnnotations xmi:id="_C7z7gEYwEdqKKKoghK_EBg"
source="appliedStereotypes">
        <contents xmi:type="network_0:network__WorkStation"
xmi:id="_C7z7gUYwEdqKKKoghK_EBg" ip="xx.xx.xx.1"/>
      </eAnnotations>
    </ownedMember>
```

5.5.2. How ModelBus supports UML2 profiles.

5.5.2.1. Objectives

1. ModelBus should enable the tools to exchange profiled models. In other words, a consumer tool can send and receive profiled models to the provider tool as modeling service parameters.
2. The tag values that are attached to the profiled models should be correctly transmitted. No information is lost during transmission.

3. ModelBus should offer three mechanisms for transmitting profiles models.
 - a. Transmitting the profile definitions with the model. This mechanism is useful when the model owner tool own the profile definitions that do not exist in the model receiver tool.
 - b. Transmitting only the model without the profile definitions. This mechanism is useful when the model receiver tool already own the profile definitions. Avoiding the transmission of the profiles tunes up the performance.
 - c. Mixture of both mechanisms. Some profiles are transmitted with the models and the other profiles are not transmitted. This mechanism offers the advantages of both previous ones.

5.5.3. API usage

5.5.3.1. *Choosing profiles to be transmitted or not to be transmitted*

We assume that a model is an EMF resource. This EMF resource refers to other EMF resources represented profiles. We call them the “**model resource**” and the “**profile resources**”.

ModelBus allows the developers to specify which profile resources need to be transmitted and which profile resources do not need to be transmitted. To do so, the developer configures the default serializer (see the class `org.eclipse.mddi.modelbus.adapter.infrastructure.serialize.emf_xmi.DefaultModelSerializer` and the method `getIgnoredUriPrefixSet()`) to skip transmitting the resources with the URI beginning with the specified prefixes.

By default, the serializer skip all the transmission of resources having the prefixes : “pathmap://ModelBus” , and “http://” . The resources beginning with those prefixes are supposed to be present at the receiver side (therefore, no transmission is required).

The class `ProfiledModelSerializer` (in the package `org.eclipse.mddi.modelbus.adapter.infrastructure.serialize.uml2`), is a specialization of the default serialization. It particularly deals with the UML2 profiled models. It is initially configured to skip, (in addition to the “pathmap://ModelBus” and “<http://>” prefixes), the profile resources beginning with following prefixes :

- “pathmap://UML2_PROFILES”
- “pathmap://UML2_LIBRARIES”
- “pathmap://UML2_METAMODELS”

For flexibility, this configuration can be modified by users as follows.

```

Serializer s = new DefaultSerializer(); // or ProfiledModelSerializer()

s.getIgnoredUriPrefixSet().removeAll();

s.getIgnoredUriPrefixSet().add("newPrefix"); // .....

adapterStub.getTransportManager().setSerializer(s);

```

Therefore, the users can specify themselves that they want to skip transmitting resources with URIs beginning with which prefixes.

5.5.3.2. *How to configure the profile resources not to be transmitted.*

In order to use profiled model transmission, the provider and consumer tools are required to perform following steps.

Consumer Tool

Register profiles: Register the profile definitions

Provider Tool

Register profiles: Register the profile definitions

that are manipulated by tools

that are manipulated by tools

Load a profiled model: Load the profiled model to be transmitted as input parameters in the memory (if not yet loaded)

Deploy Adapter

Invoke the modeling service.

Performs the service execution

Receive the service result

The steps written in bold require the uses of specific ModelBus Adapter API for profile manipulation.

5.5.3.2.1. *Register profiles*

ModelBus avoids transmitting profile definitions with models for optimizing performance. ModelBus Profile registration enables ModelBus to correctly reconstruct profiled models upon the reception of models.

The API for registering a profile definition is illustrated below. The classes and methods in bold are provided by ModelBus.

```
String umlResourceURI =
"jar:file:/D:/eclipse3.1.1/plugins/org.eclipse.uml2.resources_1.1.0.jar!/";
Uml2ModelUtil.init(URI.createURI(umlResourceURI));

// for each Profile definition
Profile profile = Uml2ModelUtil.registerProfile(
URI.createFileURI("src/org/eclipse/mddi/modelbus/adapter/test/profile/MyProfile.
profile.uml2"));
```

The **umlResourceURI** refers to the jar file provided by the Eclipse UML2 tool. It contains the metamodel UML2, the definition of basic primitive types, and the definition of basic profiles.

The parameter of the method **Uml2ModelUtil.registerProfile(URI uri)** is the URI reference to the profile definition XMI file. ModelBus will load this profile definition from this URI. In the case that your tool already has the UML profile loaded in memory, it is possible to use the method **Uml2ModelUtil.registerProfile(Profile profile)**, whereas the parameter is the Profile object that has already been loaded.

5.5.3.2.2. *Load a profiled model*

If the tool already has the profiled model loaded in the memory, this step can be skipped. However, the tool integrator must make sure that the following conditions are satisfied for the loaded profiled model.

Link between a profiled model and the profile definition: The profiled model must be correctly linked to the profile. In EMF, an unresolved link is represented as a proxy object. If you try to perform `System.out.print` the Profile object, to which is linked your profiled model is linked, and see the `eProxyURI` tag (e.g. `rg.eclipse.uml2.impl.ProfileImpl@26dbec (eProxyURI: MyProfile.profile.uml2 #_9PAf0EYmEdqKKKoghK_EBg)`), then your model is not correctly link to the profile.

No duplication of the Profile object. The Profile object, to which the profiled model is linked, must be the same object that is registered.

In order to verify both conditions, we provides the method **boolean Uml2ModelUtil.hasLink(org.eclipse.uml2.Model model, org.eclipse.uml2 .Profile profile)**. It returns true of the model is correctly link to the profile. The tool integrator can call this method with the profiled model he wants to transmit as service parameter and the Profile object that has previously been registered. If it returns false, it is possible that there are duplications of the Profile objects in the system.

In order to facilitate tool integration task, we provide an API for loading a profiled model from an XMI file for transmitting this model through ModelBus. The loading of a profiled model consists of the following steps.

- As previously described, the XMI file of the profiled model references the profile definition with a URI. The tool integrator needs to inform ModelBus that this URI refers the Profile object that has previously been registered. So that, when loading the profiled model, ModelBus will link this model with the Profile object that is already loaded. This step enables ModelBus to avoid the duplicated loading of the Profile object. The tool integrator invoke the method `Uml2ModelUtil.setAliasURIToProfile(Uri uri, Profile profile)`. “uri” is the URI that refers to the profile definition XMI file and that is contained in the profiled model XMI file. “profile” is the Profile object that has previously been registered.
- Next, the tool integrator can load the profiled model using the method `Collection Xmi2EmfConversion.loadFromFile(Uri uri)`, where “uri” refers to the profiled model XMI file. This method returns the collection of the top-level objects in this XMI file. Generally, a UML model XMI file contains one top-level object of type `org.eclipse.uml2.Model`.
- The last step is just for verifying the two conditions that we described. `Uml2ModelUtil.hasLink(...)` should return true.

```
Uml2ModelUtil.setAliasURIToProfile(Uri.createFileUri("MyProfile.profile.uml2"),
profile);

Model model = (Model) Xmi2EmfConversion.loadFromFile(
"src/org/eclipse/mddi/modelbus/adapters/test/profile/MySystem.uml2").get(0);

// test link between the model and the profile
Uml2ModelUtil.hasLink(model, profile);
```

5.5.4. Test case

To validate our approach, we have provided the following tool integration test case. Our test case consists of a service that receives a profiled model and returns the modified profiled model. The correctness of transmission is verified by examining the tag values of the transmitted model: 1) the provider tool prints the tag values of the model received the consumer tool. 2) the consumer tool prints the tag values of the result model received from the provider tool.

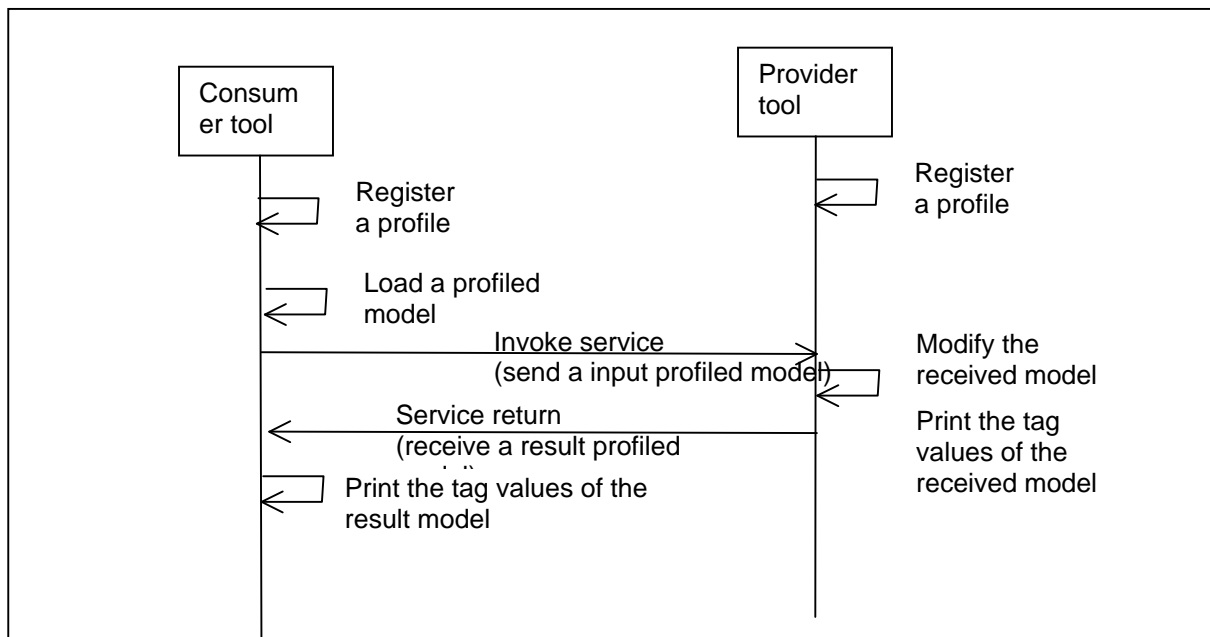


Figure 3 Test case

5.5.5. Internal mechanism

This section describes the implementation techniques of ModelBus. It serves for the developers who contribute to the ModelBus platform.

5.5.5.1. Overview

We employ the following techniques for transmitting a profiled model.

A global URI is assigned to the registered profile. A profile with the name "Profile1" will have the global URI "pathmap://ModelBus/UML2Profile/Profile1.profile.uml2". By using this rule, each profile will have a unique global URI, independently from the file location. If the provider and consumer tools use the same profile, then they will agree on the same global URI of this profile.

When a profiled model is transmitted, the links the profile definition is encoded as the global URL. The profile definition is not transmitted with the model.

Upon reception, when ModelBus deserializes the profiled model, it looks up for the Profile object corresponding the global URI transmitted with this profiled model. Then ModelBus links the profiled model with the Profile object.

5.5.5.2. *Managing a set of registered profiles.*

Class `ModelBusResourceSet.GlobalResourceRegistry` manages a map from global URIs to profile definitions. It provides the following methods.

```
Resource getResource(Uri uri)
void registerResource(Uri globalURI, Resource resource)
```

5.5.5.3. *Serializing a profiled model by encoding with the global URI of the profile*

In EMF, when serializing a model that consists of multiple resources, you have to serialize resource by resource. When a resource is serialized, it refers to the other resources with the URI.

A profiled model consists of at least two resources: the one of the model (*model resource*) and the one of the profile definition (*profile resource*). To serialize the profiled model for transmission, we do the followings

First, we set the URI of the profile resource to the global URI [`profile.eResource().setURI(globalURI)`].

Then, we serialize the model resource [`model.eResource().save(outputStream, null)`]. The serialized model will refer to the profile with the global URI.

This mechanism is implemented in class `Emf2XmiConversion`.

5.5.5.4. *Deserializing a profiled model.*

A `ResourceSet` is used for managing the saving (serializing) and the loading (deserializing) of a set of inter-related resource. We provide an implementation of `ResourceSet` that extends the default implementation, in order to take into account global URIs. The class `ModelBusResourceSet` enables the deserializing of the profiled model with the following mechanism.

`ModelBusResourceSet` overrides the method `Resource delegatedGetResource(Uri uri, boolean loadOnDemand)`. This method is invoked to resolve a uri to the resource. When the `ModelBus Adapter` receives a profiled model, it uses `ModelBusResourceSet` to deserialize this model. During the deserialization, the method `delegatedGetResource(...)` will be called back in order to resolve a global URI to a profile definition. Once it is called, `ModelBusResourceSet` looks up for the registered profile in the `GlobalResourceRegistry`. Then, the resolved profile object is linked to be deserialized profiled model.

6. Acronyms / Dictionary

Context – is part of URL. For servlet containers (e.g. Tomcat), “context” means location of the application relatively to container root.

Local Call (Local Invocation) – this term defines a transport mechanism, which is applied for communication (calls, invocation) between tools situated in the same JVM class loader.

Qualified Service Name – in all operation that requires a modelling service name, this name can be qualified. This means that it is possible to fully specify the service by the Tool name, Interface name and Modelling Service name. In this case modelling service name is delimited by “.”. For example, for OCLCheck service the qualified name can be “OSLO.interface1.OCLCheck”

Relative Path – a part of URLpath used to identify End Point (e.g. modeling _services/testTool or notification/SampleNotificationConsumer)

URL – Universal Resource Location, for example “http://localhost:8081/modelbus/modeling _services/testTool”. In the frame of ModelBus URL represents an Web Service address by which a modelling service or notification service can be accessed. URL is compose of protocol identifier (e.g. http://), host (e.g. 127.0.0.1 or localhost), port (e.g. :8081), and URLpath (e.g. /modelbus/notification).

URLpath – a part of URL. URLpath is composed of context (e.g. /modelbus) and relative path(e.g. modeling _services/testTool)

7. References

- [REF 1] ModelWare Project Proposal – Annex 1 “Description of Work”.
- [REF 2] ModelBus Software Architecture Documentation volume I
<http://www.eclipse.org/mddi/D3.1%20ModelBus%20Architecture%20Specification%20-%20Volume%20I.pdf>
- [REF 3] ModelBus Software Architecture Documentation volume II
<http://www.eclipse.org/mddi/D3.1%20ModelBus%20Architecture%20Specification%20-%20Volume%20II.pdf>
- [REF 4] ModelBus Development Plan.
- [REF 5] Eclipse MDDi site <http://www.eclipse.org/mddi/>
- [REF 6] Tool Integration Guide: using Adapter stubs
- [REF 7] Eclipse Plug-in Development Environment
http://www.eclipse.org/documentation/pdf/org.eclipse.pde.doc.user_3.0.1.pdf
- [REF 8] EMF <http://www.eclipse.org/emf/>
- [REF 9] Axis version 1.2 <http://ws.apache.org/axis/releases.html>

8. License Agreement

ModelBus is released under Eclipse Foundation Public License, which can be found hereafter.

Eclipse Public License - v 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and

b) in the case of each subsequent Contributor:

i) changes to the Program, and

ii) additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents " mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

2. GRANT OF RIGHTS

a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.

b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

a) it complies with the terms and conditions of this Agreement; and

b) its license agreement:

i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

iv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

- a) it must be made available under this Agreement; and
- b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR

IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. The Eclipse Foundation is the initial Agreement Steward. The Eclipse Foundation may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.