ARCHER Developer Guide

METADATA CREATION PACKAGE

Version: 1.1
Date: 2008-08-22
Status: Release

Change History

Version	*Date*	*Author*	*Description*
0.1D	2008-01-29	Abdul Alabri	First Draft
0.2D	2008-01-30	Ron Chernich	Format and Introduction
0.3D	2008-04-23	Ron Chernich	Update screen shots; add logging section.
1.0D	2008-04-30	Steve Crawley	Rewrote chapter on schemas; add section on record reqs, etc.
1.0	2008-05-01	Steve Crawley	MDE 1.0 Release
1.1	2008-08-22	Ron Chernich	Revise for 1.1 release

Table of Contents

1. Introduction

1.1. Purpose of Document

The Metadata Editor (MDE) has been designed as an application-neutral facility for the creation and editing of arbitrary name-value pair based metadata that is based on specific metadata schemas. This document describes the actions a software developer must undertake in order to incorporating the Metadata Editor into an Application.

1.2. Scope

The audience for this document is Software Developers and Integration Specialists who require understanding of the detailed procedures for the use of the MDE as a "black box" service. Operational details of the editor are not included (refer to the User Manual for this information).

1.3. Background

This package was designed and built to maintain high quality metadata records guaranteed to conform to a specific metadata schema at a nominated version. It reflects over a decade of experience with earlier tools and incorporates feedback from the largely non-technical user community. Until recently, the features that users have demanded have necessitated an editor implemented as a thick client, desk-top application. The service described in this manual leverages what has been labelled "Web 2.0" technology to provide a light-weight, browser-based client that is responsive and flexible. The server side components have been designed to be self-contained with a simple HTTP based API that enables the package to be integrated with all commonly used web programming languages.

1.4. Context

The MDE is implemented as a Java web service that must be deployed in a container such as Tomcat. For persistence of metadata records (load and save) the editor relies on a Service Provider Interface (SPI) implementation that must be written by the application developer.

1.5. Terms and Abbreviations

Apache Tomcat	*a* commonly used, open source, web services container.
Application	a collection of tools which are used collectively to achieve a business process - roughly analogous to a program, as a normal user perceives it.
Hover-text	a system of non-interactive help that causes a window containing explanatory text to automatically appear id the client mouse pointer pauses in a specific UI location for a preset time.
Integration Specialist	*a* person trained and having responsibility for the setup and maintenance of development, testing, deployment, and possibly other web server configurations.

Metadata data describing data.

Widget a graphic user interface component

Acronym	*Expansion*
CGI	Common Gateway Interface.
EMF	Eclipse Modelling Framework
HTML	Hypertext Mark-up Language
HTTP	Hypertext Transfer Protocol
J2EE	Java 2 Enterprise Edition
JAR	Java Archive
JDK	Java Development Kit
JNDI	Java Naming and Directory Service
JSON	JavaScript Object Notation
JSP	Java Servlet Pages
MCP	Metadata Creation Package
MDE	Metadata Editor
MDSR	Metadata Schema Repository
MMF	Metadata Management Facility
MSE	Metadata Schema Editor
MSF	Metadata Schema Facility
MSS	Metadata Schema Schema
SPI	Server Provider Interface
UI	User Interface
URI	Universal Resource Identifier
URL	Universal Resource Locator
WAR	Web Application Archive
XML	eXtensible Mark-up Language

Table 1

1.6. References

MCP-FRS *ARCHER Functional Specification – Metadata Creation Package,* Version 1.3 (Draft), 2007-07-13

EMF-DG *Eclipse Modeling Framework: A Developer's Guide,* Addison-Wesley, 2003, ISBN 0-13-142542-0.

ExtJS-2.0 See: http://extjs.com

1.7. Domain Knowledge

- Familiarity at the user level with the World Wide Web (WWW) is assumed.
- Integrators are expected to be familiar with configuring services under *Apache Tomcat*.

2. Integration

2.1. Introduction

This section is intended for web server and container Integration Specialists responsible for making the central components available to software developers and for creating production environments. Integration Specialists do not need a detailed understanding of components and processes described in the other sections of this Development Guide. The distribution package contains the components listed in Table 2.

Name	Purpose
`mde.zip`	The zip file containing the content from which the MDE implementation WAR file is assembled.
`archer-client-metaservice.jar`	Used in development environments to resolve the SPI references. This jar need not and should not be included in the application WAR file as the classes are globally available from the web server `common/lib` loader.
`archer-client-metaservice.jar`	This archive must be copied to the web server `common/lib` directory and an entry made for JNDI as detailed in this guide.
org.json.jar	This archive must be copied to the web server `common/lib` directory.
org.eclipse.*.jar	These archives must be copied to the web server `common/lib` directory.
au.edu.archer.metadata.msf.mss_*.jar	These archives must be copied to the web server `common/lib` directory.

Table 2

This section assumes that the web services container used is *Apache Tomcat*.

2.2. Deployment

The deployment process requires the following steps:

1. Deploy the common archives (JAR files)
2. Configure the Record Broker in the JNDI
3. Deploy and name the MDE servlet (WAR file)

4. Configure the names selected for the Record Broker and MCP servlet in applications that will use them.

2.2.1. JAR Distribution

The Java Archive (JAR) file containing the Record Broker and associated classes must be made globally available by copying it to the `$CATALINA_HOME/common/lib` location.

The associated SPI library (`archer-client-metaservice.jar`) may be distributed to Archer Application developers to resolve interface references during development. It should *not* be included in application Web Archive (WAR) packages as the interfaces and exceptions are made globally available as described above.

2.2.2. JNDI Configuration

Systems Integrators have the responsibility for determining the name by which the Record Broker factory will be globally known. This must be configured into the Apache Tomcat web service configuration file (`$CATALINA_HOME/config/context.xml`) to include a reference to the Object Factory. An example configuration is shown in Figure 1. The value for the *name* attribute is arbitrary, but must be used by all MDE aware applications. For optimum maintainability, it should be provided by indirection from an entry included in each application deployment descriptor. This allows it to be changed without chasing through code and performing re-compilations.

```
<Context>
  ...

  <Resource name="bean/RecordBrokerFactory"
            auth="Container"
            type="au.edu.archer.spi.MetadataServiceProvider"
            factory="au.edu.archer.recordbroker.RecordBrokerFactory"
/>

</Context>
```

Figure 1

2.2.3. Servlet Configuration and Deployment

The WAR file (`mde.war`) that contains the MDE serrvlets must be configured before it can be used. The MDE distribution provides the WAR file contents as a compressed (zip) file, together with a Perl script called `Configure.PL` that will configure and assemble the WAR file. This script performs a number of tasks including:

- setting the MDE context items in the “web.xml” deployment descriptor,
- changing logging properties such as level, log size, etc.,
- incorporating external metadata schemas (for a “flat-file” schema repository), and
- removing test data for deploying a “production” MDE.

The configuration utility follows the standard Unix “configure” conventions. There is more information on the utility's options in

the MDE release notes, but for the most current option descriptions, run the utility with the `--help` option (or no options):

```
./Configure.PL --help
```

After configuration, the `mde.war` file must be copied to the Apache Tomcat `$CATALINA_HOME/server/webapps` directory[1].

The WAR files contains a deployment descriptor file that sets the relative URL by which the service can be invoked by Applications. If it is necessary to change this or other parameters not covered by the `Configure.PL` script, you can do the following:

1. manually unpack the WAR file using the Java `jar` command (this requires a Java SDK installation),
2. use a text editor to modify the deployment descriptor and / or make other changes as required, and
3. use the Java `jar` command to repackage the WAR file ready for deployment.

2.2.4. Application Configuration

This process will depend on the individual Applications as detailed in the specific documentation packages. A well written Application should have a configuration descriptor that permits integration specialists to configure the JNDI name for the Record Broker and the URI name (full, or relative) for the deployed MDE servlet without editing, recompilation, and packaging.

2.3. Dependencies and Environment

The MCP service has been validated under the following environments:

- Java JRE 1.5.x, or later
- Apache Tomcat version 5.5.x
- Linux RHEL 5 (also tested under FC5, 6, and 7)
- MS Windows XP Pro
- Microsoft Internet Explorer 7
- Mozilla Firefox 2.x

1 *Depending on how your Tomcat server is set up, you may also need to manually remove the `$CATALINA_HOME/server/webapps/mde` directory tree and restart Tomcat.*

3. Editor Architecture

3.1. Overview

The Metadata Editor is a rich User Interface (UI) web application based on Ext JS 2.0 JavaScript library. When the Metadata Editor is loaded, the UI is created and populated with the elements that are defined in the record, using a representation of the metadata schema specified by the record to populate lists and other editor features. Metadata elements and attributes in the record are bound to appropriate UI widgets. For example, a metadata record containing the tag `<comment>A comment.</comment>` will be rendered in the UI as a *Text Box* containing the editable text, "A comment.", with a label that identifies the element type as being a "comment". Currently, the Text Box will appear in schema definition order. Figure 2 shows an example of a metadata record rendered for editing. Full details of editor functionality are provided in the MDE User Guide which is included with the distribution package, and can be downloaded from the editor itself via the `Help` menu.

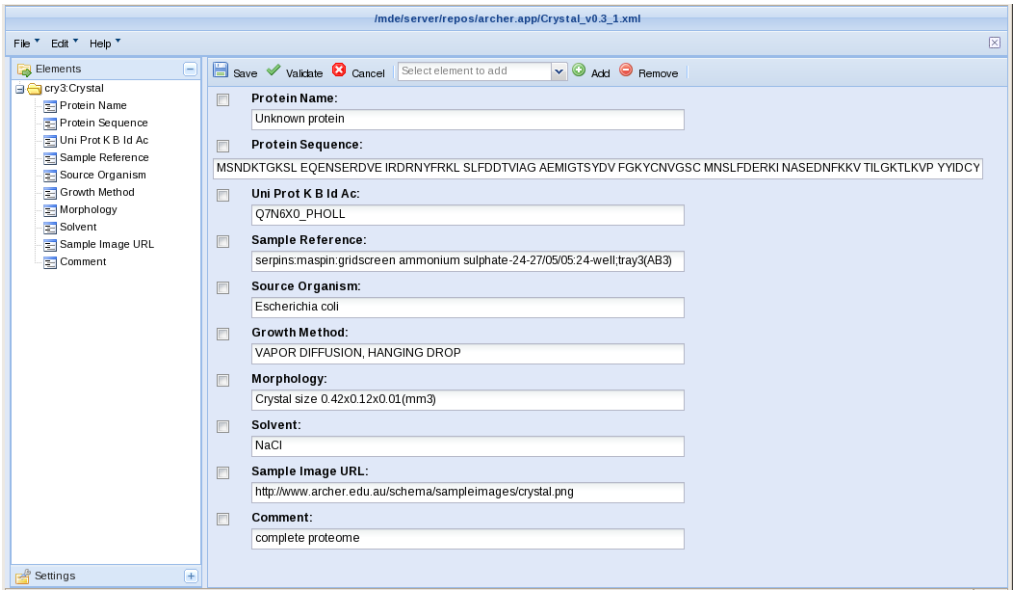

Figure 2

The Metadata Editor loads and persists records through integration with application dependant implementations of the Service Provider Interface (SPI). These must be written by the Application developers. The SPI defines the common wrapper placed around heterogeneous metadata repositories that Archer Applications must implement to provide Metadata Editor with read/write access to the underlying records. Currently, the SPI defines the following methods:

- **Bind**: Saves a record. This method generates a new Id and binds it with the given record. The Id must not already be in use.
- **Rebind**: Update a metadata record. If the given record Id already exists, the record is replaced otherwise it is inserted.
- **Lookup**: Retrieve a metadata record by Id if it exists.

- **Remove**: Delete a metadata record from the repository

Currently, only the `rebind and lookup` methods are used by the editor and related batch processing facility. A simple example implementation is shown in Appendix D - Sample SPI Implementation.

While the editor imposes no restriction on how records are stored within an Application, they must be supplied to the editor through the SPI lookup method as XML documents. These documents need to obey certain rules in order to be viewable as metadata records using MDE, as described in the next section.

3.1.1. Limitations

The Metadata Editor does not provide a means for deleting records. As this action may have referential issues associated with it, this is seen as a responsibility of the application invoking the editor.

The editor does not create new records. If this function is required, the Application should create an empty record so that its ID can be passed to the editor for element value entry.

The editor cannot create new versions of an existing record, i.e. there is no *"save-as"* capability.

3.2. Metadata Record Requirements

The Metadata Editor is not an editor for arbitrary XML. Rather, an XML document needs to satisfy a number of requirements before it can even be successfully opened by MDE. These requirements are partly militated by the requirements of generic XML parsers in Java and Javascripts, and partly by the needs of MDE. In the latter case, the primary issue is that MDE needs to be identify a single definitive MSS metadata schema (see later) for each record.

An example of an XML document satisfying these requirements (and also conforming to the schema in Appendix C – Sample MSS Schema) is given in Appendix B – Sample Metadata Record.

3.2.1. Well-formed XML

A metadata record must be well-formed XML according to the XML 1.0 or 1.1 specification. For example, '<' and '>' characters should appear in the right places, element tags should be balanced, attributes should be properly quoted and so on.

3.2.2. Defined XML Namespace

The root element of the metadata record must have a well-formed XML namespace declaration or declarations. The first declared (i.e. default) namespace is used to resolve the record's metadata schema.

3.2.3. XML Schema Location

The root element of the metadata record must also have a well-formed `xsi:schemaLocation` attribute. A schema location attribute defines a list of pairs that map namespace URIs to schema location URIs. In addition to

being well-formed, MDE requires that the list includes a mapping for the XML document's default namespace[2].

3.2.4. Known Schema

The previous requirements allow MDE to go from a record's default namespace to a schema location URI. In addition, the schema location URI must match an MSS metadata schema that is available to MDE using the following algorithm:

1. The URI is parsed into its components, and everything apart from the path component is ignored.
2. The directory path is removed, leaving just the local name.
3. If the local name has the suffix ".xsd", that is removed.
4. What remains is used as the schema key. The MDE then attempts to lookup the MSS form of the schema using that key.

3.2.5. Correct Root Element

The MSS schema specifies many things that a "valid" metadata record needs to conform to. However, one particular property must be satisfied before MDE can start to validate a record.

The MSS schema includes a component called the XML Schema Profile which specifies how metadata elements are mapped to XML elements. This profile specifies an XML namespace URI and a root element for the metadata record. If the actual record does not contain exactly one XML element whose namespace and name correspond to the XML schema profile, the record cannot be validated.

3.2.6. XSD Schema Conformance

The final requirement is that any XML record should conform to the XSD that MSF generates from the MSS form of the schema.

MDE does not specifically check XSD conformance, and in most cases you can get away with non-conformant XML. However, there are cases where MDE will behave incorrectly when presented with a non-conformant record.

3.3. Invoking the Editor

The Metadata Editor is launched by a HTTP request to the URL configured during integration for the editor servlet. It is *strongly* recommended that this URL not be hard-coded, but read from a configuration file allowing the ARCHER Integration team to configure the namespace as they wish. Record and work-flow related details for the editor call are supplied as mandatory and optional CGI parameters. An example call is shown in Figure 3. Once invoked, the Metadata Editor will be loaded in a new browser window, retaining the content of the call in the original window. Optional call parameters allow the user to return to the calling application with out losing the current session, or to invoke a new URL.

2 *In fact the current MDE implementation insists that <u>every</u> namespace URI that is mentioned in the schema location attribute is a declared namespace. This is arguable a bug: the XML specifications do not require this, and it is certainly not needed to make MDE schema resolution work.*

```
<a href =”/mde/mde.jsp?record=123&
          next=/myapp/next.jsp&
          cancel=/myapp/cancel.jsp”>Edit Metadata</a>
```

Figure 3

3.4. CGI Parameters

The CGI parameters that are required to launch the editor are listed in Table 3. Applications that link the Metadata Editor should be tested to ensure that they supply valid values for these parameters before launching the Metadata Editor.

Parameter	Obligation	Description
`debug`	optional	To set the debug level
`record` †	mandatory	The identifier of a record to be edited. Passed to the user written `MetadataStoreProvider` implementation to load and save records.
`template` †	optional	Document identifier recognized by the `MetadataStoreProvider` implementation as a template to create a new record (blank or partially composed).
`title`	optional	Editor window page title
`next`	optional	The forwarding URL when user closes the editor
`cancel`	optional	The forwarding URL on user initiated cancel
`readOnly`	optional	If “true”, then the document cannot be edited or saved (Disables field editing, Save, Add Element, and Remove Element actions).

Table 3

† Mutually exclusive.

3.5. Logging

For convenience, the MDE maintains its own logging using the Tomcat version of the JDK logging package. The logging properties are specified in the `logging.properties` file packaged in the top level of the deployed MDE war file. The usual configurable items for log level, log size, append, etc are provided. These may be configured using the `Configure.PL` script provided in the distribution, or changed manually following deployment. By default. The log file(s) will be written to:

```
$CATALINA_HOME/webapps/mde/logs
```

The MDE will not start if it is unable to commence logging. In this event, an error message will be written to the Catalina logs.

Note: it has been found that logging is not properly configured if you launch Tomcat 5.5.x using the `jsvc` utility according to the instructions in the Tomcat 5.5 online documentation. It is recommended that you start Tomcat using either:

```
$CATALINA_HOME/bin/startup.sh
```

or

```
$CATALINA_HOME/bin/catalina.sh start
```

4. The Service Provider Interface

4.1. Introduction

The editor architecture is centred around a `Record Broker`. This object is responsible for loading and saving metadata records using Application provided keys. The Record Broker itself is obtained from an Object Factory that is globally bound into the web server Java Naming and Directory Interface (JNDI). The Record Broker implements the `MetadataServiceProvider` interface. Users interact through this interface alone allowing the underlying implementation to change without need for recompilation.

Every application using the MDE must provide an implementation of the `MetadataStoreProvider` interface. This object must be registered with the Record Broker during application initialization. In most cases, the implementation will be a trivial wrapping of the existing metadata repository. An application may register as many implementations as required. Each implementation must provide a unique identifier string during registration. The Record Broker will determine which store to use to service a read/write request from the record identifier. This string is provided by the application when the editor is invoked and must commence with the identifier under which the responsible persistence wrapper was registered. The remainder of the record key allows the persistence layer to identify the required record.

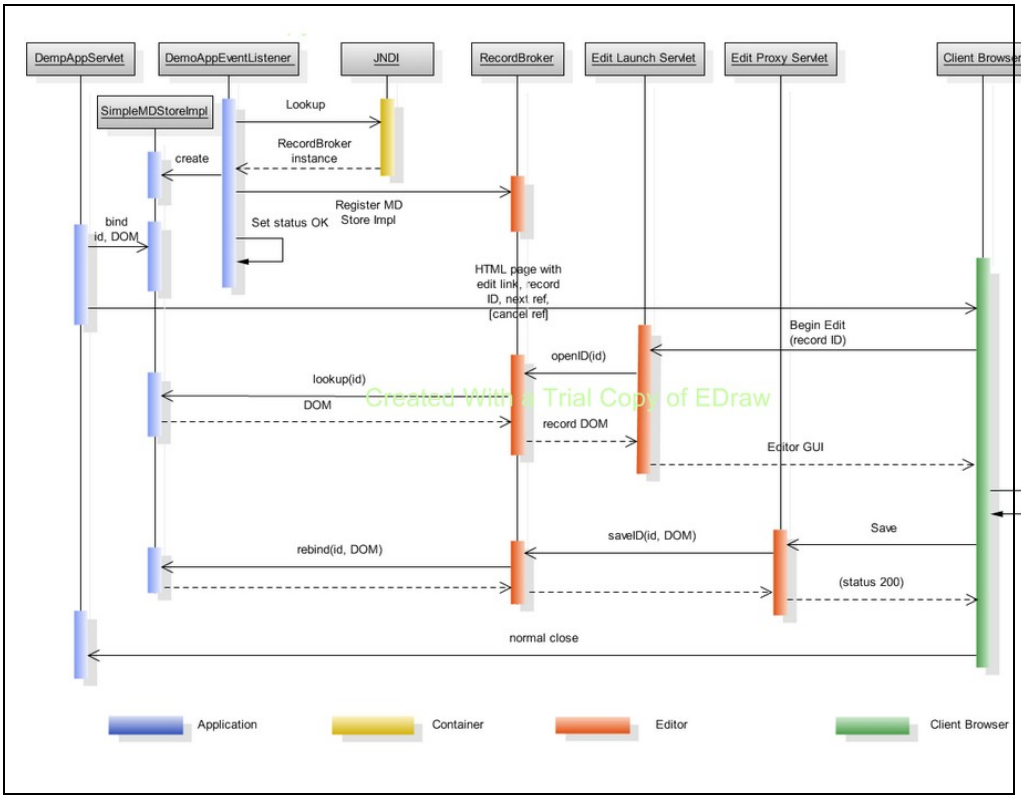

Figure 4

Figure 4 depicts the object interactions for Application start-up and later editing of a metadata record. For convenience, the Application is depicted

as a classic J2EE Servlet. Actual implementations may use JSP or other technologies able to interact with the web services container to register their `MetadataStoreProvider` interface implementation with the `RecordBroker`.

4.2. Implementation

The Service Provider Interface (SPI) defines an interface that must be implemented by all Applications wishing to use the MCP. The individual implementations provide a common view over all of the applications' underlying metadata persistence mechanisms.

Developers must implement the `MetadataStoreProvider` interface allowing the MCP to load and persist metadata records (see Appendix A – SPI Interface). The interface itself is patterned after the CORBA naming service (as is the JNDI). Note that only the `rebind()`, and `lookup()` methods need to be implemented. The others are currently included for historic reasons and may be deprecated in the final release unless found beneficial by application developers.

The implementation class must be registered with the Record Broker during Application start-up. When the MCP requests load or save of a record, it passes the URL of the record to the Record Broker which locates the Provider class responsible for that record using the host part of the record ID URL. The implementation then assumes responsibility for:

- Retrieving the record from storage and translating the local native record format to XML (for `lookup()` operations).
- Translating a XML record to native format and writing it to persistent storage (for `rebind()` operations).
- Performing any record locking operations required to provide adequate concurrency.

5. Schema Preparation and Configuration

5.1. Introduction

As should be clear from earlier sections of this document (and from other MCP documents), metadata schemas play a central role in the Metadata Creation Package.

This section discusses how metadata schemas are represented, how they are created, and how they are managed. Most of this is implemented by Metadata Management Facility (MMF) code that is separate from the MCP deliverables. In fact, MMF is currently split into two distinct components:

- the Metadata Schema Facility (MSF) is used for creating metadata schemas, and
- the Metadata Schema Repository (MDSR) which provides a shared repository for "published" schema.

5.2. Metadata Schemas

The primary purpose of a metadata schema is to specify what constitutes a "valid" or "acceptable" metadata record. The schema specifies what fields are valid, and how many times they are required or allowed to occur. It specifies what the field types are, and in some cases constrains the field values beyond simple data types.

Based on the metadata schema, MCP will:

- map XML elements to metadata elements,
- build an MDE screen layout to display the elements, complete with user-friendly labels and description as hover-text,
- select and configure the MDE data entry widgets to do initial validation on data entry, and
- perform a thorough record validation on demand; e.g. when saving a metadata record.

In the initial MCP documents it was envisaged that MDE would use the XML Schema Language (XSL aka XSD) to specify metadata schemas, and that MDE would support a subset of XSL that was appropriate for metadata records. It became apparent that this approach was problematic, so a decision was made to design a project-specific language for representing metadata schemas. This language has come to be known as MSS.

Some of the key features of the Metadata Schema language (MSS) are:

- separation of metadata information and representation models,
- provision for rich document, including multi-lingual element names and descriptions,
- a rich language for expressing constraints,
- specification completeness; i.e. no reliance on external XSDs, and
- it has an XML-based concrete syntax, like XSL and DTD.

MSS and the accompanying Metadata Schema Editor (MSE) comprise the Metadata Schema Facility (MSF). These are described in detail in the "Metadata Schema Facility User Guide".

5.3. Schema Creation

Metadata schemas for MCP are specified in Metadata Schema language and represented as ".mss" files. Like the XSL, MSS uses an XML-based concrete syntax. Unlike XSL, MSS is defined in terms of a meta-model. In fact, the meta-modelling framework we are using (EMF) can generate the vast majority of the code needed for model editing and XML serialization code "for free".

To simplify the task of creating MSS schemas, the Metadata Schema Facility includes an Metadata Schema Editor (MSE). MSE's functionality includes:

- display and editing of metadata schemas,
- validation of metadata schemas to pick up obvious inconsistencies,
- creation of shared type definitions for reuse across multiple schemas,
- generation of "flattened" metadata schemas for use by MCP,
- generation of equivalent XSL schemas, and
- export of schema files to a Metadata Schema Repository (MDSR) instance.

The output from the MSF consists of metadata schemas represented as text files in MSS and XSD format. These can either be uploaded to a shared schema repository (e.g. MDSR) or embedded into the MCP as described later.

MSE is implemented in Java as a set of plug-ins that run within the Eclipse RCP framework[3]. This means that MSE must installed as a desktop application. The instructions for doing this are included in the MSF README file.

5.4. The Metadata Schema Repository

The second part of the MMF is the Metadata Schema Repository (MDSR). While it is possible (as explained later) to configure MDE with "hard-wired" metadata schemas, it is better to fetch schemas from a shared repository. This approach ensures that MDE installations (and other applications) can all use the same schema definitions, and it eliminates the problem of propagating schema updates to multiple places. A fully fledged repository also allows users to examine schemas and look for alternatives.

As the name suggests, the MDSR is a web-based service for storing metadata schemas. The primary functions of MDSR are as follows:

- providing secure storage for schemas with multiple representation formats and multiple versions,
- supporting HTTP requests to upload and retrieve schemas, and
- providing a web-based user interface for browsing and managing the schema repository.

Like MDE, the MDSR is implemented as Java servlets that run under Tomcat 5.5. MDSR uses Fedora for persistent storage of schema files. The MSDR is described in detail in the MDSR User Guide and associated documentation.

If you are going to use an MDE with MDSR, you must first populate the repository with the relevant schemas. The easy way to do this is with MSE.

3 *In fact, the MSE editor is mostly generated from the MSS meta-model.*

The following instructions assume that you have previously installed MDSR and MSF/MSE according to the relevant instructions. They also assume that you have been provided with the MSS format schemas as archive:

1. Start the MDSR instance on the server if it is not already running; see the MDSR documentation.
2. Launch an MSE instance on the machine where it is installed; see the MSE documentation.
3. In MSE, go to the workbench and use the “File” menu to create an empty Project.
4. Select the Project, and use the context (right-button) menu to Import the Archive file into the Project.
5. For each MSS model file that represents a schema do the following:
    a. Double-click the file in the “Navigator” window to open the file in the MSS model editor.
    b. Expand the top node of the tree view and select the Metadata Schema element.
    c. From the context menu, run the export wizard by choosing “Export MDSR”.
    d. The first pane should say that the model is valid. Click “Next>”
    e. The second pane asks for the 'hostname' and 'port number' for the MDSR. Make sure that the values in the field are correct, then click “Next>”.
    f. The third pane allows you to set the schema key, add a log message and control what gets uploaded. You should be able to just click “Finish”.
    g. You should next see a progress dialog, followed by a dialog that says that generation has succeeded. Click “OK”.
    h. Finally should see another progress dialog, followed by a dialog that says that the schema has been uploaded. Click “OK”.

If you have the relevant “flattened” MSS schema, it is also possible to “ingest” and “data stream” them to the MDSR without using MSE. For each schema you will need to:

1. Use “Ingest Object” to register the schema, setting the “Object Identifier” field to the schema key that MCP uses to identify the schema; e.g. “DataCollection_v0.4”. Note that there is no suffix on the schema key!
2. Use “Add/Edit Datastream” to upload the “flattened” MSS schema file. Make sure that you select the right “Object Identifier” and that you enter “MSS-Flat” as the “Datastream Id”.
3. You can also upload the non-flattened MSS file and the XSD file if you want, but MCP currently does not use them.

5.5. Configuring MDE to use an MDSR

When an MDSR instance has been set up and populated, the final step is to configure the MDE servlets to fetch their schemas from the MDSR. This is done using the `Configure.PL` script described earlier:

```
./Configure.PL –-with-mdsr-repos \
      –-with-mdsr-host <host> --with-mdsr-port <port>
```

If the `–-with-mdsr-host` and `–-with-mdsr-port` options are omitted, they default to `localhost` and `8080` respectively.

Once this has been done, MDE users will be able to use new (or updated) schemas added to the MDSR with no reconfiguration to the MDE servlets.

There is a small time window (currently 3 minutes) during which an MDE will use an old cached version of a schema[4]. And (of course!) schema changes will not affect MDE editor windows opened before the change was made.

5.6. Creating and Configuring a "flat file" Schema Repository

While the use of MDSR is recommended, it is also possible to configure MDE to use a built-in "flat-file" repository. The "flat-file" repository makes use of copies of schema files that have been configured into the MDE WAR file and deployed to the Tomcat container.

The MDE distribution includes a default set of MSS schemas, comprising versions of the Xtal schemas defined by Bob Cameron and a couple of test schemas. If you run the `Configure.PL` script as follows:

```
./Configure.PL –-with-ff-repos
```

If (as is likely) you need to use a different set of schemas, you first need to prepare a directory containing the schemas. Specifically, the schemas to be included must have a name of the form `<key>-flattened.mss`, where `<key>` is the schema key that MDE will use to denote the schema and version. Now you need to run the `Configure.PL` script as follows:

```
./Configure.PL –-with-ff-repos –with-ff-schema-path <dir>
```

where `<dir>` is the name of the directory containing the schema files.

Of course, if you use the "flat-file" repository approach, you will need to repeat this procedure *for every MDE installation* each time you need to add or update schemas.

4 *To be specific, the MDE caches schemas that it fetches from the MDSR for 3 minutes. This is done to try to reduce the chance that MDE requests will overwhelm an MDSR instance. The 3 minute caching interval is a compromise between load management and timely propagation of changes. In the long term, cache lifetimes should be tunable, possibly from the MDSR side (as in DNS).*

Appendix A – SPI Interface

CAUTION! This code is provided for reference only and may differ from the actual release. For the most up to date version, see the current release package.

```
/*
 * Copyright (C) 2007  School of Information Technology and Electrical
 * Engineering, University of Queensland (www.itee.uq.edu.au).
 * This program was developed as part of the ARCHER project (Australian
 * (Research Enabling Environment) for the Department of Education
 * Science and Training.
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

package au.edu.archer.metadata.spi;

import org.w3c.dom.Document;


/**
 * Defines the common wrapper placed around heterogeneous metadata repositories
 * that Archer Applications must implement to provide Metadata Editor read/write
 * access to the underlying records.
 * @author xchernich
 */
public interface MetadataStoreProvider
{
  /**
   * Persists the passed record.
   * @param metarecord Metadata in DOM form.
   * @return The unique, repository dependant identifier for the persisted record.
   * @throws MetadataProviderException Unable to persist record
   */
  String bind (Document metarecord)
    throws MetadataProviderException;

  /**
   * Persists the passed record using the passed key. If a record with
   * the key already exists, it is silently replaced. If it does not,
   * it may be inserted.
   * @param metarecord DOM metadata
   * @param key Unique identifier for record (repository dependant)
   * @throws MetadataProviderException Unable to persist record
   */
  void rebind (Document metarecord, String key)
    throws MetadataProviderException;

  /**
   * Retrieve the matadata in DOM form for the provided identifier.
   * @param key Unique identifier, repository dependant.
   * @return Metadata DOM for application specific key, or null
   * @throws MetadataProviderException Invalid or unknown key
   */
  Document lookup (String key)
    throws MetadataProviderException;

  /**
   * Removes the record with the passed key. Implementations may
   * choose not to implement this method.  It is expected this
   * method will be deprecated in the final release.
```

```
   * @param key Unique identifier, repository dependant.
   * @throws MetadataProviderException Invalid or unknown key
   */
  void remove (String key)
    throws MetadataProviderException;

  /**
   * Returns a list of all identifiers in the repository. Implementations may
   * choose not to implement this method.  It is expected this
   * method will be deprecated in the final release.
   * @return All known identifiers
   */
  String [] list ();

}

// EOF
```

Appendix B – Sample XML Metadata Record

```
<?xml version="1.0" encoding="utf-8"?>

<dcol:DataCollection
  xmlns:dcol="http://www.archer.edu.au/schema/xsd/datacollection/0.3/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.archer.edu.au/schema/xsd/datacollection/0.3/
file:DataCollection_v0.3.xsd">

     <dcol:numberOfFrames>180</dcol:numberOfFrames>
     <dcol:numberOfFrames>180</dcol:numberOfFrames>
     <dcol:crystalTemperature>298.0 give or take a bit</dcol:crystalTemperature>
     <dcol:angleIncrement>0.5</dcol:angleIncrement>
     <dcol:angleRange>90.0</dcol:angleRange>
     <dcol:totalRotationRange>90</dcol:totalRotationRange>
     <dcol:probe>x-ray</dcol:probe>
     <dcol:radiationType>CuK\a</dcol:radiationType>
     <dcol:diffractionSource>rotating weasel</dcol:diffractionSource>
     <dcol:detectorType>area detector</dcol:detectorType>
     <dcol:resolution>2.0</dcol:resolution>
     <dcol:collectionDate>2007-02-04</dcol:collectionDate>
     <dcol:xrayWavelength>1.5418</dcol:xrayWavelength>
     <dcol:detectorDistance>200.0</dcol:detectorDistance>
     <dcol:exposureTime>10.0033334</dcol:exposureTime>
     <dcol:detectorName>RAXIS</dcol:detectorName>
     <dcol:detectorVersion>6.X, c++</dcol:detectorVersion>

</dcol:DataCollection>
```

Appendix C – Sample MSS Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<mss:MetadataSchema xmi:version="2.0"
        xmlns:xmi="http://www.omg.org/XMI"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:mss="http://au.edu.archer.metadata.msf/mss/1.0"
        name="DataCollection"
        uri="http://archer.edu.au/metadata/DataCollection"
        version="0.4">
  <description>This metadata records details about the device and methods used to
collect the diffraction images</description>
  <label>Data Collection</label>
  <elements name="collectionType" uri="" minOccurrences="1" maxOccurrences="1">
    <description>The type of image collection.</description>
    <label>Collection Type</label>
    <type xsi:type="mss:ControlledList">
      <terms value="Native"/>
      <terms value="Derivative"/>
    </type>
  </elements>
  <elements name="solutionMethod" minOccurrences="1">
    <description>The method(s) used to determine the structure.</description>
    <label>Solution Method</label>
    <type xsi:type="mss:ControlledList">
      <terms value="SAD"/>
      <terms value="MAD"/>
      <terms value="SIR"/>
      <terms value="SIRAS"/>
      <terms value="MIR"/>
      <terms value="MIRAS"/>
      <terms value="ISAS"/>
      <terms value="ISAR"/>
      <terms value="ISARAS"/>
      <terms value="AB INITIO" uri=""/>
      <terms value="DM"/>
      <terms value="MR" uri=""/>
    </type>
  </elements>
  <elements name="numberOfFrames" maxOccurrences="1">
    <description>The total number of frames in the scan.</description>
    <label>Number Of Frames</label>
    <type xsi:type="mss:ConstrainedType">
      <baseType xsi:type="mss:IntegerType"/>
      <constraints name="IntegerMustBeNonnegative">
        <expression xsi:type="mss:Operation" operator="GreaterOrEqual">
          <operands xsi:type="mss:Self"/>
          <operands xsi:type="mss:Literal" value="0"/>
        </expression>
        <errorMessage>This integer value must be >= zero</errorMessage>
      </constraints>
    </type>
  </elements>
  <elements name="crystalTemperature" maxOccurrences="1">
    <description>The temperature of the crystal during collection in degrees
Celsius.</description>
    <label>Crystal Temperature</label>
    <type xsi:type="mss:DecimalType"/>
  </elements>
  <elements name="angleIncrement" maxOccurrences="1">
    <description>The angular distance between the start angles of two consecutive
images in the scan (in degrees).</description>
    <label>Angle Increment</label>
    <type xsi:type="mss:DecimalType"/>
  </elements>
  <elements name="totalRotationRange" maxOccurrences="1">
    <description>The total number of degrees scanned.</description>
    <label>Total Range Rotation</label>
    <type xsi:type="mss:DecimalType"/>
  </elements>
  <elements name="radiationType" maxOccurrences="1">
    <description>Name of the type of radiation used.</description>
    <label>Radiation Type</label>
    <type xsi:type="mss:ControlledList">
      <terms value="x-ray"/>
```

```
      <terms value="electron"/>
      <terms value="neutron"/>
      <terms value="gamma"/>
    </type>
  </elements>
  <elements name="radiationNature" maxOccurrences="1">
    <description>The nature of the radiation. This is typically a description of the
X-ray wavelength in Siegbahn notation.</description>
    <label>Radiation Nature</label>
    <type xsi:type="mss:StringType" maxLength="50"/>
  </elements>
  <elements name="diffractionSource" maxOccurrences="1">
    <description>The source type of the radiation.</description>
    <label>Diffraction Source</label>
    <type xsi:type="mss:ControlledList">
      <terms value="rotating anode"/>
      <terms value="synchrotron"/>
      <terms value="sealed tube"/>
    </type>
  </elements>
  <elements name="detectorType" maxOccurrences="1">
    <description>The general class of the radiation detector.</description>
    <type xsi:type="mss:ControlledList">
      <terms value="area detector"/>
      <terms value="ccd detector"/>
    </type>
  </elements>
  <elements name="highResolution" uri="" maxOccurrences="1">
    <description>The smallest value for the interplanar spacings for the reflection
data in angstroms.</description>
    <label>High Resolution</label>
    <type xsi:type="mss:DecimalType"/>
  </elements>
  <elements name="collectionDate" maxOccurrences="1">
    <description>The date of the start of the scan.</description>
    <label>Collection Date</label>
    <type xsi:type="mss:DateType"/>
  </elements>
  <elements name="wavelength" maxOccurrences="1">
    <description>The wavelength of the radiation in angstroms.</description>
    <label>Wavelength</label>
    <type xsi:type="mss:DecimalType"/>
  </elements>
  <elements name="detectorDistance" maxOccurrences="1">
    <description>The distance from the sample to the detector along the beam in
millimetres.</description>
    <label>Detector Distance</label>
    <type xsi:type="mss:DecimalType"/>
  </elements>
  <elements name="exposureTime" maxOccurrences="1">
    <description>The exposure time to the beam in seconds.</description>
    <label>Exposure Time</label>
    <type xsi:type="mss:DecimalType"/>
  </elements>
  <elements name="detectorName" maxOccurrences="1">
    <description>The make, model or name of the detector device used.</description>
    <label>Detector Name</label>
    <type xsi:type="mss:StringType" maxLength="50"/>
  </elements>
  <elements name="detectorVersion" maxOccurrences="1">
    <description>The version identifier of the detector device used.</description>
    <label>Detector Version</label>
    <type xsi:type="mss:StringType" maxLength="50"/>
  </elements>
  <profiles xsi:type="mss:XMLSchemaProfile"
uri="http://www.archer.edu.au/schema/xsd/datacollection/0.4/" schema="/" tag="dcol">
    <imports tag="dcterms" namespace="http://purl.org/dc/terms/"
schemaLocation="http://dublincore.org/schemas/xmls/qdc/2006/01/06/dcterms.xsd"/>
    <layout>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.0"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.1"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.2"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.3"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.4"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.5"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.6"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.7"/>
```

```
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.8"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.9"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.10"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.11"
xmlTypeName="dcterms:W3CDTF"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.12"/>
      <parts xsi:type="mss:XMLElementLayout" xmlName="" element="//@elements.13"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.14"/>
      <parts xsi:type="mss:XMLElementLayout" element="//@elements.15"/>
      <parts xsi:type="mss:XMLElementLayout" xmlName="" element="//@elements.16"/>
    </layout>
  </profiles>
</mss:MetadataSchema>
```

Appendix D - Sample SPI Implementation

```
/**
 * Demonstrates how an Archer Application might implement the metadata record
 * persistance wrapper over their repository so that the Archer metadata Editor
 * can load and persist records.
 *
 * @author xchernich, alabri
 */
public class SimpleMDStoreImpl implements MetadataStoreProvider
{
    private static final SimpleMDStoreImpl m_instance = new SimpleMDStoreImpl();

    private static final Hashtable<String, Document> m_store =
            new Hashtable<String, Document>();

    private static int m_nextKey = 1000;

    /**
     * Provider is implemented as a Singleton that keeps all metadata in a
     * (static) hash table keyed by record ID (without the application unique
     * string)
     *
     * @return The Instance.
     */
    public static final SimpleMDStoreImpl getInstance()
    {
        return m_instance;
    }

    /**
     * (non-Javadoc)
     *
     * @see au.edu.archer.spi.MetadataStoreProvider#bind(org.w3c.dom.Document)
     */
    public String bind(Document metarecord)
           throws MetadataProviderException
    {
        String key = null;
        synchronized (m_store) {
            key = new Integer(m_nextKey++).toString();
            m_store.put(key, metarecord);
        }
        return key;
    }

    /**
     * (non-Javadoc)
     *
     * @see au.edu.archer.spi.MetadataStoreProvider#rebind(org.w3c.dom.Document,
     *      java.lang.String)
     */
    public void rebind(Document metarecord, String key)
           throws MetadataProviderException
    {
        if ((key != null) /* && (m_store.containsKey(key)) */) {
            synchronized (m_store) {
                m_store.put(key, metarecord);
            }
        }
    }

    /**
     * (non-Javadoc)
     *
     * @see au.edu.archer.spi.MetadataStoreProvider#lookup(java.lang.String)
     */
    public Document lookup(String key)
           throws MetadataProviderException
    {
        if (!m_store.containsKey(key)) {
            throw new MetadataProviderException("Key not found '" + key + "'");
```

```
        }
        return m_store.get(key);
    }

    /**
     * (non-Javadoc)
     *
     * @see au.edu.archer.spi.MetadataStoreProvider#remove(java.lang.String)
     */
    public void remove(String key)
           throws MetadataProviderException
    {
        if (!m_store.containsKey(key)) {
            throw new MetadataProviderException("Key not found '" + key + "'");
        }
        m_store.remove(key);
    }

    /**
     * (non-Javadoc)
     *
     * @see au.edu.archer.spi.MetadataStoreProvider#list()
     */
    public String[] list()
    {
        synchronized (m_store) {
            Set<String> keys = m_store.keySet();
            String[] tmp = new String[keys.size()];
            keys.toArray(tmp);
            return tmp;
        }
    }

}

// EOF
```