# Table of Contents

**Introduction** .............................................................................................................. 4

**Glossary of terms** ......................................................................................................... 4

**Document Structure** .................................................................................................... 4

**Using the XMerge Conversion Framework** .................................................................. 5

- **Introduction** .................................................................................................................. 5
- **Registry** .......................................................................................................................... 5
- **ConverterFactory** ......................................................................................................... 5
- **Convert, Document and ConvertData** .......................................................................... 6
- **Merging** .......................................................................................................................... 6

**Framework Overview** .................................................................................................... 8

- **Container Classes** ........................................................................................................ 8
  - Document........................................................................................................................... 8
  - ConvertData..................................................................................................................... 8

**Managing Plugins** ......................................................................................................... 9

- **What is a Plugin?** .......................................................................................................... 9
- **Plugin Registration** ....................................................................................................... 10
  - ConvertInfoMgr................................................................................................................. 10
  - ConverterInfo.................................................................................................................. 10
  - ConverterInfoReader....................................................................................................... 10
- **Retrieving Plugins** ....................................................................................................... 10
  - ConverterFactory............................................................................................................ 11
  - Converter.................................................................................................................... 11

**Developing a Plugin** ...................................................................................................... 12

**Overview** ...................................................................................................................... 12

**The Plugin Configuration XML File** ............................................................................ 12

- **Elements of a Plugin Configuration XML File** ........................................................... 12
  - Root Node ....................................................................................................................... 12
  - Converter Description ................................................................................................. 13
  - Converter-display-name ............................................................................................... 13
  - Converter-description .................................................................................................. 13
  - Converter-vendor ......................................................................................................... 13
  - Converter-class-impl .................................................................................................... 13
  - Converter-xslt-serialize .............................................................................................. 14
  - Converter-xslt-deserialize ......................................................................................... 14
  - Converter-target ....................................................................................................... 14

**Plugin SPI** ..................................................................................................................... 15

- **PluginFactory** .............................................................................................................. 16
  - DocumentSerializerFactory ......................................................................................... 16
  - DocumentDeserializerFactory .................................................................................... 16
  - DocumentMergerFactory ............................................................................................. 16
- **Converter SPI** ............................................................................................................. 17
  - DocumentSerializer .................................................................................................... 17
  - DocumentDeserializer ............................................................................................... 17
- **Merge SPI** ................................................................................................................... 18
  - DocumentMergerFactory ............................................................................................. 18
  - Difference Generation ................................................................................................. 18
  - Iterators ....................................................................................................................... 18
  - ConverterCapabilities ............................................................................................... 18
Introduction

StarOffice and OpenOffice are well-known for the wide variety of high-quality filters, in particular for the Microsoft Office formats.

The XMerge SDK provides a Java-based framework for converting documents between different formats as well as a growing set of conversion plugins to read and write each format. Currently HTML and the StarOffice/OpenOffice Text XML document format are provided, other formats will be released shortly.

The goal of the SDK is to provide a convenient Java API for applications that wish to convert between formats and to support developers who want to integrate support for their document formats into StarOffice and OpenOffice.

In addition to format conversion, the XMerge SDK provides a framework for merging changes in one document format (usually a simpler or lower quality format) into an original document (typically in a richer format). The benefit of merging is that richer styles or content in the original document are retained, but the edits made to the simpler format can be applied or merged into the original document.

To meet these goals there are two significant interfaces in XMerge:

- the Conversion API’s, used by applications to use a supported format conversion
- the Plugin SPI, which is used by developers who want to add conversion support for a new document format

Glossary of terms

- **Format** – a document format such as StarWriter, HTML, Microsoft Word
- **Conversion** – a translation from one document format to another
- **Merging** – applying changes made to a simple document format to an original richer document formats
- **Lossy** – a conversion to a simpler document format that loses some layout, style or even content information
- **Plugin** – a provider or adaptor that supports a specific document format
- **XMerge Registry** – a collection of declarative meta-data about document formats and the supported capabilities; each format is described by an XML document within each jar file containing a plugin for a document format

Document Structure

The document is divided into sections for developers with different needs:

- **Using the XMerge Conversion Framework** describes how to write an application that uses the highest-level API within the XMerge SDK to easily perform format conversion
- **Framework Overview** provides a description of the architecture of the XMerge conversion framework
- **Developing a Plugin** describes how to implement a document format plugin that integrates into the XMerge framework
Using the XMerge Conversion Framework

Introduction

A client interacts with three Java interfaces to access conversion and merge utilities provided by the Small
Device framework.

• An interface for registering plug-ins that perform conversion and/or merge functionality.
• An interface for converting between StarOffice XML format and a small device format.
• An interface for merging changes from the small device format back into an original StarOffice XML
document.

These interfaces have been designed to work in a generic fashion, so that the framework is utilized in the
same fashion regardless of the formats and/or plug-ins being used.

Registry

All plug-ins must have an associated converter.xml file that describes its capabilities. This converter.xml file
is represented in the client by an instance of the ConverterInfo object. A single converter.xml file may
specify the details for more than one plug-in.

When a plug-in is provided in a jar file, it is expected that the jar file will contain the converter.xml file in the
META-INF directory. As a convenience the ConverterInfoReader class has been provided to read in a
given jar file (assuming it contains a META-INF/converter.xml file), and provide a Vector of
ConverterInfo objects.

Once an instance of the ConverterInfo object has been created (via the ConverterInfoReader or
otherwise), it may be registered with the ConverterInfoMgr, which is responsible for keeping track of all
plug-in registration. The ConverterInfoMgr is a singleton class, so only one instance of it may exist in a
given JVM.

The following logic would be used for registering a specific plug-in.

```java
ConverterInfoReader cir = new ConverterInfoReader(
    "file:///path_to_jar/myPlugIn.jar",
    false);
Enumeration jarInfoEnumeration = cir.getConverterInfoEnumeration();
ConverterInfoMgr.addPlugIn(jarInfoEnumeration);
```

Example 1: Registering a plugin

ConverterFactory

The ConverterFactory provides an instance of the appropriate Convert class for a particular
conversion. The desired conversion can be specified in one of two ways:

1. As a "from-mime-type" and "to-mime-type" combination.
2. By specifying the specific ConverterInfo that describes a plug-in.
The Convert class is capable of converting a document from one specified format to another. A Document is how a single file is managed within the framework. Different plug-ins may use different implementations of the Document interface to support the needs of the plug-in. The OfficeDocument implementation is provided for Documents that use the StarOffice XML format.

The Document.read method is called to set the Document to a specific InputStream. The Document.write method is called to write the Document to a client-supplied OutputStream. The Convert method provides a getOfficeDocument and getDeviceDocument methods so that the client can easily access the appropriate Document types to be used for a given conversion.

The ConvertData class is used as the mechanism for passing one or more Document objects in and out of the Convert class. This is useful in conversions that can create multiple files. For example, a word processing document with an embedded image might be converted to two files, a HTML file and a separate image file (GIF or JPEG).

The Convert class manages the input ConvertData, so the client only needs to access the Convert.addInputStream method as many times as is appropriate to add the appropriate number of input Document objects.

After specifying the input in this fashion, the client calls the Convert.convert method, which returns a ConvertData object containing the converted Document objects. The client can iterate over each output Document within the ConvertData class and write them to user-provided OutputStream object(s).

An example of interacting with the ConvertData and Convert classes follows, omitting needed try/catch blocks to make the code more readable:

```
ConverterFactory cf = new ConverterFactory();
Convert conv = cf.getConverter("staroffice/sxw","destination_mime_type");
FileInputStream fis = new FileInputStream("fileToConvert.sxw");
conv.addInputStream(processFile, fis);
ConvertData dataOut = conv.convert();
Enumeration docEnum = dataOut.getDocumentEnumeration();
while (docEnum.hasMoreElements()) {
    Document docOut = (Document)docEnum.nextElement();
    String fileName = docOut.getFileName();
    FileOutputStream fos = new FileOutputStream(fileName);
    docOut.write(fos);
    fos.flush();
    fos.close();
}
```

Example 2: Using the Framework API to convert a document

Merging

Merge is useful when an OfficeDocument is converted to a Device Document format, and the Device Document version is modified. Those changes can be merged back into the original OfficeDocument with the merger. The merger is capable of doing this even if the Device format is lossy in comparison to the OfficeDocument format.

The client accesses the merger from the Convert object via the getDocumentMerger method, which takes the original OfficeDocument in the constructor. The client must utilize the Convert.getOfficeDocument method in order to build this original OfficeDocument. Before calling the DocumentMerger.merge method, the client must first convert the device Document to an
OfficeDocument. The `DocumentMerger.merge` method is called with the device `OfficeDocument` as its argument. When the `DocumentMerger.merge` method is called, the merged output is placed in the original `OfficeDocument`.

An example of interacting with the `DocumentMerger` object follows, omitting needed try/catch blocks to make the code more readable:

```java
String mergeFile = "origDoc.sxw";
FileInputStream mergeIS   = new FileInputStream(mergeFile);
Document mergeDoc  = myConvert.getOfficeDocument(mergeFile, mergeIS);
DocumentMerger merger = myConvert.getDocumentMerger(mergeDoc);
if (merger != null) // if merger is not null, merge is supported
{
    // dataOut is the device file converted to an OfficeDocument, the
    // output of the Convert.convert method.
    Enumeration mergeEnum = dataOut.getDocumentEnumeration();
    Document converted = (Document)mergeEnum.nextElement();
    merger.merge(convertedFile);
    String fileName = converted.getFileName();
    FileOutputStream fos = new FileOutputStream(fileName);
    mergeDoc.write(fos);
    fos.flush();
    fos.close();
}
```

*Example 3: Using DocumentMerger to do a merge*
Framework Overview

The Converter framework is that part of the code that allows the client to use a simple interface to build manage and run conversions and merge(s). It also provides the mechanism by which plugins can be retrieved by the client. It was written to be as generic as possible to enable third party plugins to be written quickly and easily.

Document Handling

There are two container classes accessible in the xMerge API. The most basic container is a Document. In order to support file formats that can have multiple files in one document (e.g. HTML) the xMerge API also provides a ConvertData class. A ConvertData class consists of a Vector of Document objects and a String containing the name of the document being stored.

Document

Document is an interface consisting of the following four methods:

- void read(InputStream is)
- void write(OutputStream os)
- String getName(void)
- String getFileName(void)

Most of the implementation work for this interface is done in reading from InputStreams to some internal representation (that can be used to convert to some other format) or writing from this representation to an OutputStream. The getName method returns the name of the Document excluding the file extension, getFileName returns the name of the Document including the file extension.

ConvertData

ConvertData is a container class for multiple Document objects. This is especially useful when handling multiple files which are part of the one document. It can also be used if handling multiple documents. It has two member variables:

- Vector v
- String name

The documents are added to the Vector using the addDocument method. The Documents contained in the vector are accessed using an Enumerator returned from the getDocumentEnumeration method. Other functions used in this class are:

```
// ConvertData class

class ConvertData {
    private:
        Vector<Document> v;
        String name;

    public:
        void addDocument(Document doc) {
            // add Document to Vector
        }

        // Other methods...
};
```
The `reset` method enables reuse of a `ConvertData` object. `getName` and `setName` can be used for getting and setting the document represented by the `ConvertData`. `getNumDocuments` returns the number of documents contained in `ConvertData`.

### Managing Plugins

The object which provides the ability to register and to retrieve a plugin to enable document conversion and merger is the `ConverterFactory` class. It is through this class that `Convert` objects are returned and these can be then used to do the conversion using `ConvertData`'s.

![Figure 2: Overview of SDK](image)

### What is a Plugin?

A plugin contains classes that extends the `PluginFactory` abstract class and optionally one or more of the following interfaces: `DocumentSerializerFactory`, `DocumentDeserializerFactory` and `DocumentMergerFactory`. The plugin implements these interfaces depending on the functionality they want to provide in the plugin implementation.

A plugin also contains an XML Configuration file which describes its main classes and provides some information about the plugin. This allows plugins to be dynamically loaded by an application, without having specific knowledge of the implementation classes for that plugin beforehand.

The plugin capabilities can be discovered using introspection to determine which interfaces the plugin implements. For instance a plugin that implements the `DocumentMergerFactory` is capable of performing merges.

A plugin may implement up to seven interfaces:

- `DocumentSerializer`
- `DocumentSerializerFactory`
- `DocumentDeserializer`
- `DocumentDeserializerFactory`
- `DocumentMerger`
- `DocumentMergerFactory`
The `DocumentDeserializer` interface describes the `deserialize` method responsible for converting from a Device document to an Office document. The `DocumentSerializer` interface describes the `serialize` method responsible for converting from an Office document to a Device document. The `DocumentMerger` interface defines a merge method that is used to merge two Office documents together. The equivalent interface Factories return an instance of one of the three interfaces above.

The `PluginFactory` abstract class defines only two methods (`createOfficeDocument` and `createDeviceDocument`). The `PluginFactory` implementation however can optionally implement any of the three other interfaces (`DocumentSerializer`, `DocumentDeserializer` or `DocumentMerger`).

**Plugin Registration**

Each plugin registers itself with the framework by providing an XML document called converter.xml which describes that particular plugin's conversion and/or merge capabilities. This file must be included within each plugin jar file. The framework accesses this capabilities information using the `ConverterInfo` class.

Functionality is not accessed directly, it is provided to the client by the registry as a `ConverterInfo` class which provides a `PluginFactory` Implementation. In order for the registry to keep track of which conversions are possible each plugin must provide a converter.xml. The converter.xml is stored in the plugins jarfile. The `ConvertInfoReader` is a helper class which is capable of pulling the converter.xml out of the jarfile so it can be stored in the registry by the `ConvertInfoMgr`.

The `ConvertInfoMgr` as its name suggests manages a set of `ConverterInfo` objects. A `ConverterInfo` is simply a set of the attributes described in the converter.xml.

**ConvertInfoMgr**

The `ConvertInfoMgr` is a singleton that adds `ConverterInfo` objects to a Vector it maintains. It also has methods to return a `ConverterInfo` based on the mime types of the original and destination file formats. This is used by the `ConverterFactory` to return convert objects which will be described in Retrieving Plugins.

**ConverterInfo**

The `ConverterInfo` consists of a set of fields that describe a plugin. These fields are read in from the converter.xml. It is the Client's responsibility to add jarfiles that contain these converter.xml's although the client usually does not deal directly with `ConverterInfo` (see `ConverterInfoReader`). Apart from getters to read the various fields of the `ConverterInfo` there is also three functions to return instances of `DocumentDeserializer`, `DocumentSerializer` and `DocumentMerger`. It is these instances that the `Convert` class uses to do the requested operation.

**ConverterInfoReader**

The `ConverterInfoReader` is a helper class that can be used to read a converter.xml from a jarfile. The jarfile can be passed in using the `ConverterInfoReader` constructor along with a boolean flag indicating whether validation should be done or not. It has a method called `getConverterInfoEnumeration` that returns a `Enumeration` which can be added directly to the `ConvertInfoMgr`.
Retrieving Plugins

Once the plugins have been registered the ConverterFactory can be used to return a Convert class by simply passing in the mime types of the original and destination file formats. This Convert class can then be used to do the conversion.

ConverterFactory

The ConverterFactory is a simple class that is really just a wrapper around the ConverterInfoMgr. It provides two methods, the canConvert method which takes in two mime types and will return a boolean indicating if the conversion can be done. The second method getConverter takes in the same two mime-types and instead of returning a boolean returns a Convert class which can be used to do the conversion. There is a second implementation of getConverter which takes in a ConverterInfo and boolean (to indicate direction) instead of two strings indicating mime types.

Convert

The Convert class is returned from the ConverterFactory. It has a member variable consisting of a ConverterInfo which contains an instance of the Plugin needed for the conversion indicated in the call to getConverter in the ConverterFactory. It has an addInputStream which is given a stream and filename. It will construct a document using the plugin implementation of createDeviceDocument or createOfficeDocument depending on the direction of the conversion. This document is then added to the ConvertData and is passed to the converter implementation. It also has a getDeviceDocument and getOfficeDocument so documents can be created on the client side. The clone and reset function were added to allow reuse of the Convert object.

The most important method is obviously the convert method which will, depending on direction, use the instance of the plugin implementation contained in the ConverterInfo to call serialize or deserialize. This will return a ConvertData which can be enumerated over to return documents. The converted document can then be written to an OutputStream via the Documents write method.
Developing a Plugin

The Framework Overview section describes the main interfaces that make up a plugin.

The Plugin Configuration XML File

The Plugin Configuration file is an XML file that describes such things as the name of the PluginFactory implementation class for a plugin, as well as vendor, version and other descriptive text. When implementing a converter, the implementer must provide a Plugin Configuration file for that plugin which allows the plugin to be self-describing.

Note that the current framework does not validate the configuration files against the DTD.

A DTD for plugin configuration files is described below and the full text of the DTD appears in Appendix A.

Note that more than one conversion can be described in a Plugin Configuration file. This means that a plugin can be implemented which contains more than one implementation.

Note that it is also possible for a converter to convert to multiple target types.

Elements of a Plugin Configuration XML File

The DTD for Plugin Configuration files is relatively short and has only a few mandatory elements. Many of the elements and attributes are simply descriptive items for use by client applications.

Root Node

<table>
<thead>
<tr>
<th>XML Code</th>
<th>converters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>All converter plugin information files must have this node as their document root.</td>
</tr>
<tr>
<td>Status</td>
<td>Required</td>
</tr>
<tr>
<td>DTD</td>
<td>&lt;!ELEMENT converters (converter)+&gt;</td>
</tr>
</tbody>
</table>
Converter Description

**XML Code:** converter

**Description**
The converter node contains the description of the a conversion/merge made available within the plugin. A plugin may be able to handle more than one conversion and merge, therefore there may be more than one converter node.

The `type` attribute describes the mime type of the office document that this plugin supports.

The `version` attribute is an implementer specified version number.

**Status**
Required

There may be more than one converter node.

**DTD**
```xml
<!ATTLIST converter type CDATA #REQUIRED>  
<!ATTLIST converter version CDATA #REQUIRED>
```

Converter-display-name

**XML Code:** converter-display-name

**Description**
This is human readable string provided by the plugin implementer that provides a name for the plugin for listing/inspecting purposes.

**Status**
Required

**DTD**
```xml
<!ELEMENT converter-display-name (#PCDATA)>  
```

Converter-description

**XML Code:** converter-description

**Description**
This is human readable string provided by the plugin implementer that describes the plugin.

**Status**
Optional

**DTD**
```xml
<!ELEMENT converter-description (#PCDATA)>  
```

Converter-vendor

**XML Code:** converter-vendor

**Description**
The name of the individual implementing the converter.

**Status**
Optional

**DTD**
```xml
<!ELEMENT converter-vendor (#PCDATA)>  
```
**Converter-class-impl**

<table>
<thead>
<tr>
<th><strong>XML Code:</strong></th>
<th>converter-class-impl</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>The full name and the class that implements the PluginFactory interface for this conversion. This class is instantiated in order to get access to convert and merge functionality.</td>
</tr>
<tr>
<td><strong>Status</strong></td>
<td>Required</td>
</tr>
<tr>
<td><strong>DTD</strong></td>
<td><code>&lt;!ELEMENT converter-class-impl (#PCDATA)&gt;</code></td>
</tr>
</tbody>
</table>

**Converter-xslt-serialize**

<table>
<thead>
<tr>
<th><strong>XML Code:</strong></th>
<th>converter-xslt-serialize</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>For use with any plugin that wants to use the XSLT plugin for transforming document using XSLT. This element specifies an URL to and XSLT that is used for serializing office documents to a lossy format. NB: It is assumed that the conversion specified by converter-class-impl will make use of this XSLT.</td>
</tr>
<tr>
<td><strong>Status</strong></td>
<td>Optional</td>
</tr>
<tr>
<td><strong>DTD</strong></td>
<td><code>&lt;!ELEMENT converter-xslt-serialize (#PCDATA)&gt;</code></td>
</tr>
</tbody>
</table>

**Converter-xslt-deserialize**

<table>
<thead>
<tr>
<th><strong>XML Code:</strong></th>
<th>converter-xslt-deserialize</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>For use with any plugin that wants to use the XSLT plugin for transforming document using XSLT. This element specifies an URL to and XSLT that is used for deserializing lossy format documents to office documents. NB: It is assumed that the conversion specified by converter-class-impl will make use of this XSLT.</td>
</tr>
<tr>
<td><strong>Status</strong></td>
<td>Optional</td>
</tr>
<tr>
<td><strong>DTD</strong></td>
<td><code>&lt;!ELEMENT converter-xslt-deserialize (#PCDATA)&gt;</code></td>
</tr>
</tbody>
</table>

**Converter-target**

<table>
<thead>
<tr>
<th><strong>XML Code:</strong></th>
<th>converter-target</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>The mime-type specified in the type attribute of this node indicates that a conversion is supported between this type and the type attribute of the converter node.</td>
</tr>
<tr>
<td><strong>Status</strong></td>
<td>Required</td>
</tr>
</tbody>
</table>
| **DTD** | `<!ELEMENT converter-target EMPTY>  
<!ATTLIST converter-target type CDATA #REQUIRED>` |
Plugin SPI

The Plugin SPI consists of:

- Top level Plugin SPI interfaces
- Converter SPI
- Merge SPI

Figure 5: Plugin SPI Classes
Figure 5 shows the major interfaces and class hierarchy within the Plugin SPI (though some of the interfaces of the Merge SPI are not shown).

A plugin implementer can choose the functionality they want to provide by extending the PluginFactory interface, and then optionally implementing those interfaces they require from the Merge and Convert SPI.

### PluginFactory

*PluginFactory* is an abstract class which provides methods for creating Office and Device documents from input streams passed to the plugin. A plugin implementer must extend this class when implementing a plugin.

### DocumentSerializerFactory

This interface provides a method for returning an instance of the DocumentSerializer class. Implementing this class provides access to a means of writing a given Office document to Device specific format.

Users of the plugin can query the PluginFactory implementation class using introspection to see if this interface is implemented. If it is implemented then a specific Office to device conversion is possible.

### DocumentDeserializerFactory

Similar to the DocumentDeserializerFactory, this interface provides a factory method for returning an instance of the DocumentDeserializer class. Implementing this class provides access to a means of writing a given device document to a particular Office format.

Again, users of the plugin should use introspection to query the PluginFactory implementation class to see if this interface is implemented. If it is implemented then a the user knows that the specific device to Office document conversion is possible.

### DocumentMergerFactory

The factory method of DocumentMergerFactory - createDocumentMerger() provides access to the Merger API by returning an instance of a DocumentMerger implementation.

Again, users of the plugin should use introspection to query the PluginFactory implementation class to see if this interface is implemented. If it is implemented then a the user knows that this plugin supports merging.
Converter SPI

The Converter provides interfaces which a plugin should implement if it wants to be able to write from Office format to Device format and vice versa

DocumentSerializer

The DocumentSerializer Interface provides the serialize method for converting an Office document to a device document.

A serializer is created by calling the DocumentSerializerFactory method createDocumentSerializer which takes an instance of the Document class as a parameter. This Document object is the object to be serialized, so the plugin implementation should keep a reference to this class.

```java
ConvertData demoSerialize(DocumentSerializerFactory factory,
                          Document doc)
{
    ser = factory.createDocumentSerializer(doc);
    return ser.serialize();
}
```

*Example 4: Serialization Example*

Note that the serialize method must return its result in a ConvertData object - this is because a serialize method call may result in more than one device document being created. See ConvertData for more details.

DocumentDeserializer

The process for converting from a device format to OpenOffice format is very similar to the previous step. In this case a DocumentDeserializer is required which is acquired by calling the DocumentDeserializerFactory createDocumentDeserializer method and passing a ConvertData which contains the device documents that are to be serialized.

```java
Document demoDeserialize(DocumentSerializerFactory factory,
                         ConvertData docs)
{
    DocumentDeserializer deser =
        factory.createDocumentDeserializer(docs);
    return deser.deserialize();
}
```

*Example 5: Deserialization Example*

Calling the DocumentDeserializer.deserialize method causes the data from the device document to be read from the ConvertData object and converted into a DOM tree which is maintained internally by the Document implementation class. Calling Document.write invokes the implementation class to write out the document as an OpenOffice document.
Merge SPI

The Merge SPI provides interfaces for generating differences between two documents as well as merging those differences to create a new document. These interfaces include:

- DiffAlgorithm
- MergeAlgorithm
- Iterator

The Merge SPI is designed to merge a rich format XML file with a lossy version of that document which has been edited. The term lossy is explained below.

Some implementations of these interfaces are provided in the current framework which are suitable for many of the basic merge tasks. These are discussed later in the document.

DocumentMergerFactory

This class provides a factory method for getting a DocumentMerger instance. The instance returned by this method implements the DocumentMerger interface and this implementation class handles the details of the merge itself.

Difference Generation

In the discussion below the term "original document" refers to the original OpenOffice XML document which has been parsed into a DOM tree and which is encapsulated in an implementation class of the Converters Document interface. The term "modified document" refers to the device format document which has been deserialized to a DOM tree and is also encapsulated in a converters Document interface.

The difference process makes use of several interfaces discussed below. Several of these classes are also used in the merge process. Also, note that the output of the difference process forms the input to the merge process.

Iterators

Both the difference and the merge algorithms make use of the Converter Iterator interface. This permits the difference and merge algorithms to traverse the input documents and compare their contents in an implementation independent way.

The NodeIterator class implements the Iterator interfaces. This provides a more DOM specific iterator which can be extended to create iterators specific to particular document formats. There are currently four such classes which extend NodeIterator to provide document specific iterators when traversing the DOM tree of Writer and Calc documents. These are CellNodeIterator, ParaNodeIterator, RowIterator and TextNodeIterator. If support is required for a converter for, say, OpenOffice Presentation documents, further iterators would be required to traverse the structure of OpenOffice Presentation documents.

ConverterCapabilities

When you create an instance of a specific iterator, you have to have an instance of an object which implements the ConverterCapabilities class. ConverterCapabilities specifies which OpenOffice document tags and attributes are supported in the device document format. The ConverterCapabilities.canConvertTag and ConverterCapabilities.canConvertAttribute return either true or false depending on whether the device document can support the given tag or attribute.
The DiffAlgorithm Interface

Access to any of the difference algorithms is provided by the DiffAlgorithm interface. All difference generation algorithms should implement this interface.

The DiffAlgorithm interface has one method, computeDifferences, which takes as input two iterators - one for the original document and one for the modified document - and returns a vector of Difference object.

The Difference class encapsulates information about where a difference occurs - it reports the type of difference (add, change or delete) and where the difference arises between the original and the modified document.
For example, if the difference object holds "2" for original position, "3" for modified position and "change" for the operation, it means, that the 2nd node of the original iterator has changed into the 3rd node of the modified iterator, so we have to perform a merge between these 2 nodes.

Because the difference and merge are two separate steps, it is possible to do multiple passes of difference generation for two document from different granularity. For example, to merge two Writer based documents, a paragraph Iterator can be used first to generate differences on a paragraph basis. If there are any changes in a certain paragraph, a word Iterator can be created from that paragraph content and a word-level difference generation can then be performed.
A similar process is applied to Spreadsheet documents - difference generation can be performed first at the worksheet level, then the row level then the sheet level and so on.

The multi-pass approach allows the framework to choose the difference algorithm to apply based on the content it encounters in the document. So if we encounter a table while performing a difference on a paragraph of text, we can switch to a more appropriate difference algorithm which works better on tables.

Generating Differences

There are two basic steps required to generate the set of differences. First creates a "difftable" based on the two iterators. A difftable is an \((N \times K)\) matrix where :
- \(N = A + 1\); \(A = \) number of objects (nodes) in original iterator
- \(K = B + 1\); \(B = \) number of objects (nodes) in modified iterator

When complete, the matrix specifies the nodes that have remained the same and the nodes that have changed between the two documents.

Then generate a vector of Difference objects from this matrix. These objects represent the operations that have to be performed on the original document, to create a document that is logically equivalent to the modified document in the device.

To illustrate the process we will look at how this process is applied to the difference generation for a fictitious device document to SXW merge.
Using DiffAlgorithm to Generate Differences

```java
public Difference[] diff(Document mod, Document orig, 
ConverterCapabilities cc,)
{
    SxwDocument wdoc1 = (SxwDocument) orig;
    SxwDocument wdoc2 = (SxwDocument) mod;
    Document doc1 = wdoc1.getDocumentElement();
    Document doc2 = wdoc2.getDocumentElement();
    Iterator i1 = new ParaNodeIterator(cc, doc1.getDocumentElement());
    Iterator i2 = new ParaNodeIterator(cc, doc2.getDocumentElement());
    DiffAlgorithm diffAlgo = new IteratorLCSAlgorithm();
    // find out the paragraph level diffs
    Difference[] diffTable = diffAlgo.computeDiffs(i1, i2);
    return diffTable;
}
```

Example 6: Difference Generation

The DOM trees of both documents are retrieved and an appropriate iterator - the ParaNodeIterator class in this case - is instantiated for each DOM tree.

The iterator takes as a parameter to its constructor an instance of the ConverterCapabilities class for the specific document type. The ConverterCapabilities specify which OpenOffice document tags and attributes are supported on the device document format.

Next, an instance of an appropriate DiffAlgorithm is created, in this case the IteratorLCSAlgorithm, and the Difference vector is created. This vector forms the input to the merge.

Note that the merge algorithm needs access to the iterators used to generate the differences - they are an input to the applyDifference method call during merge.

Merge

The merge process works on the principle of taking two DOM trees and iterating over them to create a new DOM tree that represents the "most-up-to-date" version of the document. Nodes for the new merged DOM tree are selected based on the differences generated by the difference process as described above.

The present merge API supports a 2-way merge - this is when you modify a document on a device and you want to merge it back to the original OpenOffice document.

3-way merges are not currently supported. A 3-way merge is required when both the original document and the device document have been modified, and the changes from both need to be merged together.

The MergeAlgorithm Interface

The MergeAlgorithm interface has a single method, applyDifference() which takes as input two iterators - an iterator for the original document, and an iterator for the modified document - and the generated Difference vector. All merge algorithms should implement this interface. In order to carry out a merge you need to instantiate an instance of a class that implements the MergeAlgorithm interface and call its applyDifference() method.

Invoking Merge

The code fragment below illustrates the `merge()` method of the class that implements the DocumentMerger interface for a fictitious device document.
public void merge(Iterator i1, Iterator i2, ConverterCapabilities cc,)
        // merge the paragraphs
        NodeMergeAlgorithm charMerge = new CharacterBaseParagraphMerge();
        DocumentMerge docMerge = new DocumentMerge(cc_, charMerge);
        docMerge.applyDifference(i1, i2, diffTable);
    }

*Example 7: Applying Differences*

The general document merging class `DocumentMerge` is instantiated. `DocumentMerge` is an implementation of the `MergeAlgorithm` interface. This class will merge two `Document` classes. It utilizes the appropriate class that implements the `NodeMergeAlgorithm` interface - which in this case is `CharacterBaseParagraphMerge` instance. Then the `applyDifference()` method is called which takes as it's parameters the two iterators used to perform the difference run, and the difference vector resulting from the difference generation.

*Interfaces and Classes involved in a Merge*

Classes implementing the `MergeAlgorithm` are responsible for processing the `Difference` vector and finding the location where additions, deletions and merges takes place and then making the appropriate changes. Where merges are required, the class that implements `MergeAlgorithm` invokes the `NodeMergeAlgorithm.merge` method of the object that was passed to it's constructors to do the actual merging. Otherwise it manipulates DOM nodes directly using the DOM API

The merge is performed on the original document DOM tree. This implies that the `Document` instance that encapsulates this DOM tree holds the result of the merge.
Appendices

Appendix A : Lossy Conversion

When a user makes a change to a document on a device, the merge process determines the changes made and applies these changes to the unchanged original OpenOffice document. It is important to note that the original OpenOffice document is needed to determine the changes made on the device document. This allows the merge to restore features that were not available in the device application where the document was edited. Once these changes are applied, the user of the specific converter can replace the original OpenOffice document with the document which includes the edits made on the device. Note that due to unsupported features on the device, the conversion process from OpenOffice document to device document is "lossy".

An instance of lossy conversion can be seen when converting a Office Writer document to a device format that does not support lists. So – for the purposes of this example - the Converter for this document type converts each Office Writer list item to an individual paragraph. When converting back from the device format to Office Writer XML we expect to be able restore the list items by using the formatting information contained in the original Office Writer document.

However ambiguities arise when the user modifies one of the "list paragraphs" of the device document? What happens when the user adds text immediately after one of the "list paragraphs" on device? Did the user intend it to be a new list item, or a new paragraph of text. The user can make many other changes in and around the "list paragraphs" which are also ambiguous.

The only way to handle these modifications during the conversion from device format to Office Writer XML is to develop a set of heuristics where the merge tries to interpret what the user intended when editing on the device. These heuristics may accurately predict the user's intentions the majority of the time, but there will be cases when they will produce erroneous results. It may be desirable in certain circumstances to provide feedback for the user in order for them to decide what action should be taken.

Appendix B : converter.dtd

<!-- converter.dtd
Author: Brian Cameron
This DTD file is provided for documentation and development purposes, the converter does not actually validate the converter.xml files that it processes. Plug-ins will not work properly, though, if the converter.xml does not conform to this DTD specification. -->
<!-- The root node, converters, must contain one or more converter nodes, each corresponds to a converter plug-in. -->
<!ELEMENT converters (converter)+>
<!-- The converter node must contain two elements: type - The convert-from mime-type. version - The version of the plug-in. -->
<!ELEMENT converter (type, version)>
<-- Each converter node must contain these child nodes: converter-display-name - Name of the converter
converter-class-impl - The PluginFactory implementation for the plugin
converter-targets - Can be one or more of these nodes. Each contains only a "type" element. This "type" element specifies the convert-to mime-type. -->
Each converter node may contain these child nodes:

- `converter-description` - Descriptive description of the plug-in.
- `converter-vendor` - Plug-in vendor name
- `converter-xslt-serialize` - The URL of the xsl stylesheet for serialization. This stylesheet must exist if the xslt plugin implementation is to be used. It is assumed that the plug-in specified via `converter-class-impl` will make use of this value.
- `converter-xslt-deserialize` - The URL of the xsl stylesheet for deserialization. This stylesheet must exist if the xslt plugin implementation is to be used. It is assumed that the plug-in specified via `converter-class-impl` will make use of this value.

```
<!ELEMENT converter  (converter-display-name,
    converter-description?,
    converter-vendor?,
    converter-class-impl,
    converter-xslt-serialize?,
    converter-xslt-deserialize?,
    converter-target+)>
<!ATTLIST converter  type        CDATA      #REQUIRED>
<!ATTLIST converter  version     CDATA      #REQUIRED>

<!ELEMENT converter-display-name (#PCDATA)>  
<!ELEMENT converter-description  (#PCDATA)>  
<!ELEMENT converter-vendor       (#PCDATA)>  
<!ELEMENT converter-class-impl   (#PCDATA)>  
<!ELEMENT converter-xslt-serialize  (#PCDATA)>  
<!ELEMENT converter-xslt-deserialize  (#PCDATA)>  

<!ELEMENT converter-target       EMPTY>
<!ATTLIST converter-target  type  CDATA  #REQUIRED>
```
Appendix C : Debug and Test Mechanisms

The Debug class is a helper class designed to provide a mechanism for logging debug messages. It is designed as a singleton class. It can be included in a file using the following import:

```java
import org.openoffice.xmerge.util.Debug;
```

*Figure 6: Importing debugging class*

Switching on your Debug Properties

To enable your debugging settings, you must set up Java to pick up your Debug.properties file first in your Java CLASSPATH. You can do this in two ways.

- Create the directory structure org/openoffice/xmerge/util in a directory of your choosing and place your Debug.properties file there. Then add the directory to your CLASSPATH.

```bash
cd /export/home/xmerge
mkdir -p org/openoffice/xmerge/util
cp ~/Debug.properties /export/home/org/openoffice/xmerge/util/.
export CLASSPATH=/export/home/xmerge:$CLASSPATH
```

*Example 8 : Adding a Debug.properties from a directory*

- Alternatively, package your Debug.properties in a JAR file. The Debug.properties file must be in the package org.openoffice.xmerge.util within that JAR file. Then, add the JAR file containing Debug.properties to the beginning of the CLASSPATH.

Debug Properties File

There are three basic levels of debug messages that can be used Info, Error, Trace. These properties are set to either true or false in the Debug.properties file. The output an also be set to System.out, System.err or to a file using the debug.output property. Below is a sample Debug.properties file.

```properties
# Debug Logging information and settings
debug.info=true
debug.trace=true
debug.error=true
debug.output=System.err
```

*Example 9: Sample Debug.properties file*

Logging methods

There are 4 different logging methods in the debug class.

- logSystemInfo()

  This will log platform information including the operating system name version and platform as well as the JDK version and vendor.

- logTime()

  This logs a time-stamp which includes the date and the time.
• log(int, String)

This sets the debug level of the message being logged. If this level has been set to true in the Debug.properties file then the message will be logged.

• log(int, String, Throwable)

This is the same as the previous log method except it will also log a stack trace of the exception passed in.

Testing your Plugin

The Driver.java in xMerge is the client program used to test plugins. The following is the usage message displayed if you run Driver.java without any arguments

Usage:

```
java org.openoffice.xmerge.test.Driver <args>
where <args> is as follows:
-from <MIMETYPE> -to <MIMETYPE> [ -merge <OrigDoc> ] <document>
```

Figure 7: Driver Usage Message

The first thing the Driver.java does is to load the converterInfoList.properties file to read in the list of jarfiles that correspond to conversion plugins. Below is a sample converterInfoList.properties

```
#
# Sample convertInfoList.properties
#
jarname1=file:///jarDirectory/someOtherPlugin.jar
jarname2=file:///jarDirectory/myPlugin.jar
```

Example 10: Sample ConvertInfoList.properties

Once the plugins have been registered the Driver parses the command line. This consists of looping through the arguments finding the -from -to and -merge (if it is present) keywords and assigning each argument to the correct variable. These “from” and “to” mime types are what is used by ConverterFactory’s getConverter call to return a plugin to do the conversion so it important these are correct and correspond to mime types in the plugin’s converter.xml.

The next section is where the actual conversion takes place in the doConversion method. This method needs to do a lot of exception handling and error checking to return useful messages if there is a problem. The actual code to do the conversion is quite small and does the following steps.

1. Constructs a ConverterFactory and gets the Convert class with a call to getConverter.
2. Adds the file(s) to be converted by making a call to ConverterFactory’s addInputStream.
3. Makes a call to the convert() method to do the conversion and returns a ConvertData.
4. If no merge is specified, enumerates over the ConvertData writing each element out to file.
5. If merger is specified, creates a DocumentMerger and does a merge using the Document output from the conversion process. Writes out resultant Document.

The Driver should require no modification to work with any given plugin, unless there is a very specific task you want it to do. Generally, it should be enough to modify the converterinfoList.properties file to point to your plugin.