XMI Reader Component 1.0 Component Specification

1. Design

1.1 Overview

The XMI Reader component provides the ability to parse XMI files, using SAX, since using a DOM representation of a large XMI file would require too much memory. This component provides a pluggable framework for node handlers based on their element type and a repository for the elements constructed by the node handlers based on their xmi ids. For elements that are not loaded yet, a map of properties found so far is kept, until the element is actually found in the stream.

This design provides a self-configuring component (via the Config Manager component) which provides methods for parsing not only files and streams, but XMI data referenced by a URI or stored in a Zip file as well. A specialized extension of the `org.xml.sax.ContentHandler` interface is provided, geared specifically for this framework, as well as a base implementation of this interface.

1.2 Design Patterns

Delegate – The `DelegatingHandler` class does not process the parsed XMI data; it delegates to the appropriate `XMIHandler` instead.

Strategy – The various plug-in handlers implement the strategy pattern. The right strategy for each element is executed based on the element type.

Façade – The `XMIReader` class provides an easier-to-use façade than the SAX parser provides, especially for parsing from a Zip file or URI source.

1.3 Industry Standards

XML, XMI, SAX

1.4 Required Algorithms

There are no required algorithms, but this section clarifies how to implement some of the more complicated methods of the design.

1.4.1 Configuration

The one-argument constructor of `XMIReader` should use an algorithm such as this (note, exceptions are not handled in this pseudocode)

```
ConfigManager cm = ConfigManager.getInstance();
String reuseFlag = cm. getString (namespace, ”reuse”);

boolean reuse = true;
if (reuseFlag != null)
       reuse = reuseFlag.equalsIgnoreCase(“true”);

// maps from class name to XMIHandler object. They are re-used
// if the ‘reuse’ flag is set.
Map<String, Object> instantiatedClasses = new HashMap<String, Object>();

// loops through all the subproperties of the “handlers” property.
Property p = cm.getPropertyObject(namespace, "handlers");
// if the property is not found, simply returns.
Enumeration names = p.propertyNames();
```

```
while (names.hasMoreElements()) {
       // name/value pairs
       String elementType = (String) names.nextElement();
       String className = p.getValue(name);

       // create or retrieve this handler
       XMIHandler handler = null;
       boolean created = false;
       if (reuse && instantiatedClasses.containsKey(className)) {
              handler = instantiatedClasses.get(className);
       } else {
              Constructor = the default constructor of 'className'
              if the default ctor doesn't exist {
                     Constructor = the one-arg constructor that takes XMIReader
                     If that ctor doesn't exist, throw XMIReaderConfigException
              }
              handler = call the Constructor using the appropriate arguments
              if using default constructor {
                     handler.setXMIReader(this);
              }
              created = true;
       }
       // we just created it, but might want to use it in a later iteration
       if (created && reuse) instantiatedClasses.put(className, handler);

       // put into handlers map
       handlers.put(elementType, handler);
}
```

1.4.2 Parse

While the XML is being parsed, the `DelegatingHandler` will determine which, if any, `XMIHandler` should process the particular node. At two points, specifically, in `startElement` and `endElement`, the handler may change.

1.4.2.1 startElement

The `startElement` method of `DelegatingHandler` should figure out if there is a handler for the current `qName or localName (if qName is null or empty)`. If so, push it onto the `activeHandlers` stack and use it as the current handler. As described in the ZUML, the algorithm is:

```
       XMIHandler tempHandler = handlers.get(qName or localName (if qName is null
or empty));
       if tempHandler != null {
              // push the handler onto the stack
              activeHandlers.push(tempHandler)
              currentHandler = tempHandler;
       }
       if currentHandler != null
              currentHandler.startElement(. . .)
```

1.4.2.2 endElement

The `endElement` method must determine if it should pop the stack or not. Since the `startElement` method will always push if the handler is found for the `qName or localName (if qName is null or empty)`, the `endElement` method uses the same technique: pop if the handler was found for the `qName or localName (if qName is null or empty)`. But first, the `currentHandler` is used to process the `endElement`, since the `currentHandler` is still "active". As described in the ZUML, the algorithm is:

```
        if (currentHandler != null)
                currentHandler.endElement( . . .)
        // determine if we should pop the stack
        XMIHandler tempHandler = handlers.get(qName or localName (if qName is null
or empty));
        If (tempHandler != null) {
                activeHandlers.pop();
                // get current handler as the top of the stack, if it exists
                if (activeHandlers.size() > 0)
                        currentHandler = activeHandlers.peek();
                else
                        currentHandler = null;
        }
```

1.5 Component Class Overview

class XMIReader

This is the main class of the XMI Reader component. It manages a map of `XMIHandlers`, which process the XMI elements as they are parsed using a SAX parser. An XMI element name can be mapped to a single `XMIHandler`; a handler can also handle more than one element name. The `XMIHandlers` can either be set up in the configuration file (see algorithms section for details) or manually via the `add/remove/getHandler` methods. This class calls the SAX parser (in the `parse(InputSource)` method, giving it a `DelegatingHandler`, which is the `ContentHandler` actually passed into the parser. Each `XMIHandler` has access to this `XMIReader` instance. This is so that the `XMIReader` can be a central clearinghouse of the various identified elements within the XMI file, in the `'foundElements'` map. Forward references are available as well; as attributes of those forward references are established, they are updated in the `'forwardReferences'` map. When the actual object is finally found, and put into the `foundElements` map, this class will automatically remove it from the `forwardReferences` map. This class is thread safe; each method is synchronized.

interface XMIHandler extends org.xml.sax.ContentHandler

This interface represents the functionality that is required for all descendants of `ContentHandler` to implement in order to be used with the `XMIReader`. There are three methods: `setXMIReader` (so it can be used to retrieve the “found elements”), `getLastRefObject` and `getLastProperty`, both of which can be used by child node handlers to retrieve the last reference object that a parent node handler created. Implementations of this interface are required to be thread-safe.

abstract class DefaultXMIHandler implements XMIHandler extends org.xml.sax.helpers.DefaultHandler

This is a default implementation of the `XMIHandler` interface. All the methods in the interface are implemented, as well as set methods for the properties. This class does not provide any support to creating elements or resolving forward references; that is the responsibility of the specific concrete subclass. This class is thread-safe; the setter methods are all synchronized.

class DelegatingHandler extends org.xml.sax.helpers.DefaultHandler

This inner class is the actual `ContentHandler` that will be passed into the SAX parser by the `parse(InputSource)` method of `XMIReader`, the enclosing class. This class is NOT a static inner class, but a true inner class. The various methods of this class will be called by the SAX parser. Each of the methods will reference the enclosing class's "`handlers`" map to determine which `XMIHandler` implementation to call for that particular element. If there is no handler for that element, the "`currentHandler`" will be used. A stack of active handlers is also maintained, in case a child node needs information that only an enclosing node has. This class is not thread-safe; it is only called from the (synchronized) methods of the `XMIReader` class.

1.6 Component Exception Definitions

XMIReaderException extends BaseException
This is the generic exception that represents an error that occurred in any part of the XMI Reader component. It is not usually thrown; instead one of the subclass exceptions is usually thrown.

XMIReaderConfigException extends XMIReaderException
This represents an exception that occurred while configuring the XMI Reader component. This can occur during constructor (e.g., in `XMIReader's` constructor) or during parsing (which configuring the SAX parser.) It can be used to wrap an underlying ConfigManager exception or without an underlying exception.

XMIReaderParseException extends XMIReaderException
This represents an exception that occurred while parsing. It is thrown by one of the `XMIReader.parse` methods. It can be used to wrap an underlying `SAXException` or any other exception.

XMIReaderIOException extends XMIReaderException
This represents an exception that occurred while opening, reading or closing a file or stream before, during, or after parsing. It is thrown by one of the `XMIReader.parse` methods. It can be used to wrap an underlying `IOException` or any other exception.

1.7 Thread Safety

This component is thread-safe. Each method in the `XMIReader` class is marked as synchronized to ensure that only one thread at a time accesses any given object. This is necessary for two reasons:

1. SAX Parsers are never required to be thread-safe so we have to do this ourselves
2. We do not want another thread to be changing the “`handlers`” map while the “`parse`” method is running. If we allowed that, there would be inconsistent results. There are additional controls in place to ensure that the handlers map is not changed while the `XMIReader` is parsing an XMI file (specifically, by throwing `IllegalStateException` in certain methods when the parse method is active). This is required since the `XMIHandlers` will have access to the `XMIReader` in the same thread as the parsing is occurring.

In addition, since the instance of the `DelegatingHandler` class is only accessible from the (synchronized) methods in `XMIReader`, it was not necessary to also synchronize that class.

The `XMIHandler` instances are required to be implemented in a thread-safe manner. The `DefaultXMIHandler` is thread-safe since the ‘set’ methods are synchronized and the 'get' methods merely return the corresponding property.

2. Environment Requirements

2.1 Environment

- Java 1.5 is required for compilation and executing test cases.

2.2 TopCoder Software Components

- Base Exception 1.0: The base class for all custom exceptions in this component.
- Configuration Manager 2.1.5: This component is used to pre-populate the handlers map in the `XMIReader` class's constructor.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

- A SAX-compatible parser (such as Xerces, which comes with the Sun JDK) is required.

3. Installation and Configuration

3.1 Package Name

`com.topcoder.xmi.reader`

3.2 Configuration Parameters

This is a sample configuration fragment (note, all the class names are fictitious and not part of this component design).

```
<Config name="com.topcoder.xmi.reader.XMIReader">
   <Property name="reuse">
       <Value>true</Value>
   </Property>

  <Property name="handlers">
     <Property name="UML:Class">
         <Value>com.topcoder.xmi.reader.handlers.ClassHandler</Value>
     </Property>
     <!-- since reuse is true, it should not instantiate another ClassHandler. -->
     <Property name="UML:Interface">
       <Value>com.topcoder.xmi.reader.handlers.ClassHandler</Value>
     </Property>
     <Property name="UML:Stereotype">
       <Value>com.topcoder.xmi.reader.handlers.StereotypeHandler</Value>
     </Property>
  </Property>
</Config>
```

Parameter	Description	Sample values
reuse (optional)	Whether or not `XMIHandler` instances should be re-used or new instances created during configuration.	True/false. Defaults to true.
handlers (optional)	Nested property that contains name/value pairs mapping element name to `XMIHandler` class name. The names are the fully-qualified element names and the values are the fully-qualified class name for the `XMIHandler` instance that will handle	Property name: "UML:Class" Value: "`com.topcoder.xmi.reader.handlers.ClassHandler`"

	that element.	

3.3 Dependencies Configuration

Define the SAX parser's `XMLReader` instance if needed by setting the system variable "`org.xml.sax.driver`". When using the Sun 1.5 JDK, this will be set by default to the Xerces implementation that comes with the JDK. However with other JDKs it may be necessary to set this value to a particular implementation of the `XMLReader` interface.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

- Extract the component distribution
- Follow Dependencies Configuration.
- Follow the demo

4.3 Demo

4.3.1 Creating XMIReaders

There are three ways to create an `XMIReader`: empty, using the default namespace, and using a named namespace.

```
//1) creates an empty reader; empty the namespace first
ConfigManager cm = ConfigManager.getInstance();

if (cm.existsNamespace(XMIReader.DEFAULT_NAMESPACE)) {
    cm.removeNamespace(XMIReader.DEFAULT_NAMESPACE);
}

XMIReader reader = new XMIReader();

// loads the file config.xml.
// it contains two namespace: XMIReader.DEFAULT_NAMESPACE and "mynamespace"
cm.add("config.xml");

//2) creates XMIReader from default namespace
XMIReader preconfigDefault = new XMIReader();

//3) creates XMIReader from the given namespace
XMIReader preconfig = new XMIReader("mynamespace");
```

4.3.2 Managing handlers

All the XMIHandler implementations in this section are "fictional" and not part of this design. They are shown for example purposes only. The developer, when writing the demo test, should implement them as "mock" implementations in order to support the demo.

```
// this fictional class extends DefaultXMIHandler
XMIHandler classHandler = new ClassHandler();
reader.addHandler("UML:Class", classHandler);
// can be used to handle more than one type
reader.addHandler("UML:Interface", classHandler);

XMIHandler packageHandler = new PackageHandler();
reader.addHandler("UML:Package", packageHandler);

XMIHandler stereotypeHandler = new StereotypeHandler();
reader.addHandler("UML:Stereotype", stereotypeHandler);

XMIHandler operationHandler = new OperationHandler();
reader.addHandler("UML:Operation", operationHandler);


// retrieves a handler; should be ClassHandler.
System.out.println(reader.getHandler("UML:Class").getClass());

// should be InterfaceHandler
System.out.println(reader.getHandler("UML:Interface").getClass());

// removes a handler
reader.removeHandler("UML:Interface");
// should be null
System.out.println(reader.getHandler("UML:Interface"));

/*
 * The output in console:
 * class com.topcoder.xmi.reader.handlers.ClassHandler
 * class com.topcoder.xmi.reader.handlers.ClassHandler
 * null
 */
```

4.3.3 *Parsing XMI files and data*

There are four methods by which to parse XMI files: by `File` object, by `InputStream`, by URI and by Zip file.

```
File xmiFile = new File("test_files" + File.separator + "sample.xmi");
// Should parse the file, and only process class, package and
// stereotype nodes (since we configured handlers for those in 4.3.2),
// and any child nodes to those nodes. But, any other nodes
// (e.g., UML:Interface) will be ignored.
reader.parse(xmiFile);

// Suppose the PackageHandler processed this and inserted an element with this ID
// we can get the element and print it out
Object element = reader.getElement("I3fc39a51m10e2acac631mm7f49");
System.out.println(element);

// Should parse the XMI at the given URI.
reader.parse("file:///" + xmiFile.getAbsolutePath());

// Suppose the PackageHandler processed this and inserted an element with this ID
// we can get the element and print it out
element = reader.getElement("I3fc39a51m10e2acac631mm7f49");
System.out.println(element);

// does the same thing, but as an input stream.
 InputStream is = new FileInputStream(xmiFile);
 reader.parse(is);
 is.close();
 // Suppose the ClassHandler processed this and inserted an element with this ID
 // we can get the element and print it out
 element = reader.getElement("I3fc39a51m10e2acac631mm7d42");
 System.out.println(element);

 // Unzips the . zuml then parses it.
 reader.parseZipFile("test_files" + File.separator + "sample.zuml");
 // Suppose the ClassHandler processed this and inserted an element with this ID
 // we can get the element and print it out
 element = reader.getElement("I3fc39a51m10e2acac631mm7d42");
 System.out.println(element);
```

4.3.4 Managing elements

Normally, the `XMIHandler` implementations will create, then set and get the various UML model elements stored in the `XMIReader`, but they are accessible to the calling application as well.

```
reader.parseZipFile("test_files" + File.separator + "sample.zuml");
Object element  = reader.getElement("I3fc39a51m10e2acac631mm7d41");

// changes the element at this id
reader.putElement("I3fc39a51m10e2acac631mm7d41", new Object());

// removes the object at this id
reader.removeElement("I3fc39a51m10e2acac631mm7d41");
```

4.3.5 Managing element properties

During parsing, "forward references" will likely be encountered. `XMIHandler` implementations can still capture properties about elements that have not been fully defined. Specifically, the `XMIReader` can keep track of which defined elements are "waiting on" forward references. In this XMI fragment, the referenced Stereotype might not have been defined yet, but the `XMIHandler` for Stereotypes can capture the forward reference:

```
<UML:Operation xmi.id = 'I3d057933m10e2fd324d0mm7242'
             name = 'XMIReaderConfigException'
              visibility = 'public'
             isSpecification = 'false'
             ownerScope = 'instance'
             isQuery = 'false'
             concurrency = 'sequential'
             isRoot = 'false'
             isLeaf = 'false'
            isAbstract = 'false'>

        <UML:ModelElement.stereotype>
             <UML:Stereotype xmi.idref = 'I3fc39a51m10e2acac631mm7e57'/>
        </UML:ModelElement.stereotype>
...
```

To capture this, the `XMIHandler` for Stereotypes would execute:

```
reader.putElementProperty("I3fc39a51m10e2acac631mm7e57",
                           "UML:Stereotype", operationObject);
```

This connects the `operationObject` with the as-yet-undefined `UML:Stereotype` object whose id is `"I3fc39a51m10e2acac631mm7e57"`, indicating that the `UML:Stereotype` property is the property that should be used to connect the two.

Later, when the Stereotype is actually found, its XMI fragment might look like this:

```
<UML:Stereotype xmi.id = 'I3fc39a51m10e2acac631mm7e57' name = 'create'
       isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
       isAbstract = 'false'>
```

```
     <UML:Stereotype.baseClass>BehavioralFeature</UML:Stereotype.baseClass>
</UML:Stereotype>
```

The Stereotype handler could then[1] instantiate a `StereotypeObject` with all its properties, and then tell all the "waiting" objects about the newly created `StereotypeObject`.

```
       Map<String, List> waiting;
       waiting = reader.getElementProperties("I3fc39a51m10e2acac631mm7e57");
```

Then for each property, and each object on the list of that property, they can be told what the Stereotype object is. Finally, the fictional `StereotypeHandler` would actually define the Stereotype object in the reader:

```
reader.putElement("I3fc39a51m10e2acac631mm7e57", stereotypeObject);
```

which will remove the corresponding entry in the `forwardReferences` map.

5. Future Enhancements

- Use ObjectFactory to instantiate handlers from the configuration
- Use a listener model to notify handlers when elements they were waiting for have been found.
- Define an ErrorHandler for more graceful recovery from parse errors, or allow a client to configure one to handle errors
- Use the Logging Wrapper component to log parse errors.

[1] Note: this section describes the typical behavior of an `XMIHandler` implementation. The actual behavior is outside the scope of this component. Only the supporting methods of `XMIReader` are salient to this discussion.