XMI Writer - Diagram Interchange Plugin 1.0 Component Specification

1. Design

When persisting a UML design to file, the standard open file format, specified by the OMG, is XMI (XML Metadata Interchange). Part of the XMI specification includes a section where diagram information can be stored. This component is a plugin to an XMI Writer, allowing Diagram Interchange elements to be transformed to XMI and persisted.

This is achieved through a large package of classes, each of which is responsible for converting a single Diagram Interchange class into an XMI Node. These are built up recursively, mirroring the associations in the Diagram Interchange component, and handed to a Diagram2XMITransformer class. The latter acts as façade - when given any recognized diagram element, it determines which element transformer should be used, delegates the transform, and formats the resulting data.

1.1 Design Patterns

Façade: Diagram2XMITransformer acts as façade, providing a simple transformation method that hides the more complex Element-level transformation from the calling application.

Composite: The DiagramInterchangeElementTransformer instances act in a composite manner, allowing the tree of diagram information to be written in the same way as writing just a leaf of XMI.

1.2 Industry Standards

XMI - Diagram interchange. For formal specification XSD, see “Annex D: Diagram Interchange XML Schema” (pg 67) in the accompanying “Diagram_Interchange_Formal” pdf file.

1.3 Required Algorithms

The most complex part is making sure that the Element transformers abide by the XSD provided - to do this, possible code is provided in the Appendix (section 6), one part for each of the 22 element transformers in this component.
They all follow the same general structure:

a) Create the Element Node
b) Set the attributes using member values
c) Append Nodes for single members
d) Append Nodes for collection members
e) Append Nodes from the superclass node’s children
f) Return the node created in step a.

The other algorithm worth mentioning is ID resolution - there is a Sequence Diagram covering it, and in pseudo code

a) Try getting the ID for the element from the XMI Writer object
b) If it exists already, return it.
c) Otherwise, generate a new ID using Java’s UUID generator
d) Convert to a string and set the element’s ID in the XMI Writer object.
e) Return this string.

1.4 Component Class Overview

Diagram2XMITransformer:

The Diagram2XMITransformer is the communication point between the XMI Writer component, and the underlying strategies provided to transform XMI Document Interchange elements into valid XMI. The main entry point into the system is the transform(elt, stream) method, inherited from the XMITransformer interface. This takes a DocumentInterchange element and writes its XMI to an output stream. Within the component, this is done by converting the element to an XML Node first, then writing this to the stream.

DiagramInterchangeElementTransformer: << interface >>

Interface that defines the contract for specific Element transformers within the Diagram Interchange writer component. Classes that implement this interface are used to convert objects of a single type of class into their XMI representation. This is performed by the transform method, taking the element and XML document, and returning a Node of XML that conforms to the XSD for the given object. Implementations of this class are not required to be threadsafe.

PropertyTransformer:
DiagramElementTransformer:
ReferenceTransformer:
DiagramLinkTransformer:
GraphElementTransformer:
GraphNodeTransformer:
GraphEdgeTransformer:
GraphConnectorTransformer:
DiagramTransformer:
SemanticModelBridgeTransformer:
SimpleSemanticModelElementTransformer:
Uml1SemanticModelBridgeTransformer:
CoreSemanticModelBridgeTransformer:
LeafElementTransformer:
ImageTransformer:
TextElementTransformer:
DimensionTransformer:
PointTransformer:
BezierPointTransformer:
GraphicPrimitiveTransformer:
PolylineTransformer:
EllipseTransformer:

All [Class]Transfomer provides the XMI transformation strategy for instances of their corresponding [Class]. They provided the logic for the transform() method - see the appendix documentation for more details. In addition, the default no-arg constructor is provided for each. These classes does not handle their thread safety, for reasons mentioned in 1.6

1.5 Component Exception Definitions

No new components are defined - The custom UnknownElementException and XMITransformException are used from the XMI Writer component.

1.6 Thread Safety

This component is not thread safe - there is no protection against multiple threads using concurrent access to modify the state of objects in use. Instead, the XMI writing should be handled in a thread safe manner by the calling application, for example making sure

only one thread performs the writing while any others do not change the underlying data structures.

2. Environment Requirements

2.1 Environment

- Java 1.5 for compilation and development

2.2 TopCoder Software Components

- Diagram Interchange 1.0, for all the diagram interchange data structures.
- XMI Writer 1.0, for access to XMITransformer and associated exceptions.

2.3 Third Party Components

The org.w3c.dom package is required for XML parsing.

3. Installation and Configuration

3.1 Package Name

com.topcoder.uml.xmi.writer.transformers.diagram
com.topcoder.uml.xmi.writer.transformers.diagram.elementtransformers

3.2 Configuration Parameters

There is no required external configuration for this component - the only configuration step is performed programmatically, passing values into the Diagram2XMITransformer constructor. See the demo(section 4.3) for an example of what parameters to give.

3.3 Dependencies Configuration

None

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

To use the component, all that needs to be done is to construct an instance with valid element transformers, then elements of known classes can be transformed. See below for a demo of how this can be achieved.

4.3 Demo

```
// Element Transformer setup:
Map<String, DiagramInterchangeElementTransformer> transMap
    = new HashMap<String, DiagramInterchangeElementTransformer>();
String base = “com.topcoder.diagraminterchange.”;
transMap.put(base + “Property”,       new PropertyTransformer());
transMap.put(base + “DiagramElement”, new DiagramElementTransformer());
transMap.put(base + “Reference”,      new ReferenceTransformer());
```

```
// ... etc.
transMap.put(base + “Polyline”,       new PolylineTransformer());
transMap.put(base + “Ellipse”,        new EllipseTransformer());

// XML Transformer setup:
Transformer xmit = TransformerFactory.newInstance().newTransformer();
xmit.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");
xmit.setOutputProperty(OutputKeys.INDENT, "yes");

// Component initialization:
Diagram2XMITransformer trans
    = new Diagram2XMITransformer(transMap, true, xmit);
// Due to defaults, this is the same as
//    new Diagram2XMITransformer(transMap);

// use with XMI Writer:
Map<TransformerType, XMITransformer> tMap = ...; // init
tMap.put(TransformerType.Diagram, trans);
XMIWriter writer = new XMIWriter(..., tMap);
// will write the diagram data to the sample file:
writer.transform(new File(“sample.xmi”), true);

// check a few initial setups:
DiagramInterchangeElementTransformer polylineTr =
    trans.getElementTransformer(base + “Polyline);
assert( polylineTr instanceof PolylineTransformer );
try{
    trans.getElementTransformer(“unrecognisedclazz”);
} catch(UnknownElementException e){
    // this should be reached
}

// set up some elements to be transformed
//  - assume makePoint returns a Point with the given x/y
//    and makeProperty returns a property with the given key/value

// make a Diagram:
Diagram diag = new Diagram();
diag.setName(“Main Class Diagram”);
diag.setPosition( makePoint(0.0, 0.0) );
diag.setSize( makePoint(250.0, 250.0) );
diag.setViewport( makePoint(0.0, 0.0) );
diag.setZoom( 0.9 );

// make a text box to put in there, with Node container:
TextElement txt = new TextElement();
txt.setText(“Title”);
GraphNode holder = new GraphNode();
holder.addContained(txt);
holder.setPosition( makePoint(120.0, 30.0) );
holder.setSize( makePoint(40.0, 25.0) );
holder.addProperty( makeProperty(“font-size”, “20.0”) )
diag.addContained(holder); // add to diagram
```

```
// make an ellipse:
Ellipse els = new Ellipse();
els.setRadiusX(22.0);
els.setRadiusY(22.0);
els.setRotation(0.0);
els.setStartAngle(0.0);
els.setEndAngle(6.283185307179586); // 2 pi
els.setVisible(false);
els.setCenter( makePoint(200.0, 30.0) );
els.addProperty( makeProperty(“fill”, “#0000ff”) );
diag.addContained(els); // add to diagram

// transform
try{
    // see provided sample.xmi for possible output - note that
    // it it’s a complete XMI file yet, so can’t be loaded
    trans.transform(diag, System.out);
} catch( Exception e ){
    // this might occur, errors are not suppressed
}

// transform unregistered class:
try{
    trans.transform(new Integer(0), System.out);
} catch (UnknownElementException e){
    // will get here - integers don’t have a transformer
}

// transform to node:
try{
    Document doc = ...; // get document from builder
    Node node = trans.transform(diag, doc);
    assert(node.getNodeName().equals(“UML:Diagram”));
} catch( Exception e ){
    // shouldn’t get here
}

trans.setWithExceptions(false); // suppress
try{
    trans.transform(new Integer(0), System.out);
} catch (UnknownElementException e){
    // won’t get here - integers are unknown, but errors suppressed
}
```

5. Future Enhancements

Two changes that would simplify this are both more about changing how the XMI Writer overall is used, rather than this component. Firstly, it would be useful if the Node-output method of transform() was all that was required, rather than the OutputStream as is currently used. This would seemingly allow more DOM-based interactions with the transformed data, both inside the Diagram Interchange plug-in, as well as the others.

The second change would be to make the XMI Writer handle all identification. Currently, each writer plug-in is responsible for setting its own identifiers for the objects it handles - this may lead to problems if multiple plug-ins use different identification techniques, problems could also occur if they use the same and have some identifier overlap. At the very least, the duplication of effort wouldn't' be required if the XMI writer was responsible for id assignment.

Finally, if optimization is desired, it would be possible for element transformers to directly use any other transformers they require, rather than going through the façade entry. For example, a polyline could get the Point transformer from its Diagram2XMITransformer, and then *locally* format all of its waypoints, as opposed to calling the Diagram2XMITransformer's transform() each time.

6. Appendix - Transformer algorithms

PropertyTransformer:

```
Element ret = document.createElement(“UML:Property”);
ret.setAttribute("key", element.getKey());
ret.setAttribute("value", element.getValue());
```

DiagramElementTransformer:

```
Element ret = document.createElement(“UML:DiagramElement”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“visible”, “” + element.isVisible());

for(Property prop : properties){
    Element tmp = document.createElement(“UML:DiagramElement.property”);
    tmp.appendChild(diagramTransformer.transform(prop));
    ret.appendChild(tmp);
}

for(Reference ref : references){
    Element tmp = document.createElement(“UML:DiagramElement.reference”);
    tmp.appendChild(diagramTransformer.transform(ref));
    ret.appendChild(tmp);
}
```

ReferenceTransformer:

```
Element ret = document.createElement(“UML:Reference”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“isIndividualRepresentation”,  “”+element.isIndividualRepresentation());

for(DiagramElement elt : referenced){
    tmp =  document.createElement(“UML:DiagramLink.referenced”);
    Element docRef = document.createElement(“UML:DiagramElement”);
    docRef.setAttribute(“xmi.idref”, resolveID(elt));
    tmp.appendChild(docRef);
    ret.appendChild(tmp);
}
```

DiagramLinkTransformer:

```
Element ret = document.createElement(“UML:DiagramLink”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“zoom”, element.getZoom());

Element tmp = document.createElement(“UML:DiagramLink.viewport”);
tmp.appendChild(diagramTransformer.transform(element.getViewport()));
ret.appendChild(tmp);

tmp =  document.createElement(“UML:DiagramLink.diagram”);
Element docRef = document.createElement(“UML:Diagram”);
docRef.setAttribute(“xmi.idref”, resolveID(element.getDiagram()));
tmp.appendChild(docRef);
ret.appendChild(tmp);
```

GraphElementTransformer:

```
Element ret = document.createElement(“UML:GraphElement”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“visible”, “” + element.isVisible());

Element tmp = document.createElement(“UML:GraphElement.position”);
tmp.appendChild(diagramTransformer.transform(element.getPosition()));
ret.appendChild(tmp);
```

```
for(GraphConnector con : element.getAnchorages()){
    tmp = document.createElement(“UML:GraphElement.anchorage”);
    tmp.appendChild(diagramTransformer.transform(con));
    ret.appendChild(tmp);
}

tmp = document.createElement(“UML:GraphElement.contained”);
for(DiagramElement con : element.getContaineds())
    tmp.appendChild(diagramTransformer.transform(con));
ret.appendChild(tmp);

tmp = document.createElement(“UML:GraphElement.semanticModel”);
tmp.appendChild(diagramTransformer.transform(element.getSemanticModel())); // if not null
ret.appendChild(tmp);

for(DiagramLink link : element.getLinks()){
    tmp = document.createElement(“UML:GraphElement.link”);
    tmp.appendChild(diagramTransformer.transform(link));
    ret.appendChild(tmp);
}

NodeList fromParent = super.transform(element, doc).getChildNodes();
for(int i = 0; i < fromParent.getLength(); i++)
    ret.appendChild(fromParent.item(i));
```

GraphNodeTransformer:

```
Element ret = document.createElement(“UML:GraphNode”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“visible”, “” + element.isVisible());

Element tmp = document.createElement(“UML:GraphNode.size”);
tmp.appendChild(diagramTransformer.transform(element.getSize()));
ret.appendChild(tmp);

NodeList fromParent = super.transform(element, doc).getChildNodes();
for(int i = 0; i < fromParent.getLength(); i++)
    ret.appendChild(fromParent.item(i));
```

GraphEdgeTransformer:

```
Element ret = document.createElement(“UML:GraphEdge”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“visible”, “” + element.isVisible());

Element tmp = document.createElement(“UML:GraphEdge.waypoints”);
for(Point waypoint : element.getWaypoints())
    tmp.appendChild(diagramTransformer.transform(waypoint));
ret.appendChild(tmp);

for(GraphConnector con : element.getAnchors()){
    tmp = document.createElement(“UML:GraphEdge.anchor”);
    Element conRef = document.createElement(“UML:GraphConnector”);
    conRef.setAttribute(“xmi.idref”, resolveID(con));
    tmp.appendChild(conRef);
    ret.appendChild(tmp);
}

NodeList fromParent = super.transform(element, doc).getChildNodes();
for(int i = 0; i < fromParent.getLength(); i++)
    ret.appendChild(fromParent.item(i));
```

GraphConnectorTransformer:

```
Element ret = document.createElement(“UML:GraphConnector”);
ret.setAttribute(“xmi.id”, resolveID(element));

Element tmp = document.createElement(“UML:GraphConnector.position”);
tmp.appendChild(diagramTransformer.transform(element.getPosition()));
ret.appendChild(tmp);

for(GraphEdge edge : graphEdges){
    tmp = document.createElement(“UML:GraphConnector.graphEdge”);
    Element edgeRef = document.createElement(“UML:GraphEdge”);
    edgeRef.setAttribute(“xmi.idref”, resolveID(edge));
    tmp.appendChild(edgeRef);
    ret.appendChild(tmp);
}
```

DiagramTransformer:

```
Element ret = document.createElement(“UML:Diagram”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“visible”, “” + element.isVisible());
ret.setAttribute(“name”, element.getName());
ret.setAttribute(“zoom”, element.getZoom());

Element tmp = document.createElement(“UML:Diagram.viewport”);
tmp.appendChild(diagramTransformer.transform(element.getViewport()));
ret.appendChild(tmp);

Element tmp = document.createElement(“UML:Diagram.owner”);
tmp.appendChild(diagramTransformer.transform(element.getOwner()));
ret.appendChild(tmp);

for(DiagramLink link : element.getDiagramLinks()){
    tmp = document.createElement(“UML:Diagram.diagramLink”);
    Element linkref = document.createElement(“UML:DiagramLink”);
    linkRef.setAttribute(“xmi.idref”, resolveID(link));
    tmp.appendChild(linkRef);
    ret.appendChild(tmp);
}

NodeList fromParent = super.transform(element, doc).getChildNodes();
for(int i = 0; i < fromParent.getLength(); i++)
    ret.appendChild(fromParent.item(i));
```

SemanticModelBridgeTransformer:

```
Element ret = document.createElement(“UML:SemanticModelBridge”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“presentation”, element.getPresentation());
```

SimpleSemanticModelElementTransformer:

```
Element ret = document.createElement(“UML:SimpleSemanticModelBridge”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“presentation”, element.getPresentation());
ret.setAttribute(“typeinfo”, element.getTypeInfo());
```

Uml1SemanticModelBridgeTransformer:

```
Element ret = document.createElement(“UML:Uml1SemanticModelBridge”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“presentation”, element.getPresentation());

Element tmp = document.createElement(“UML:SimpleSemanticModelBridge.element”);
Element eltRef = …; // get the basic node for the contained element, using class name
```

```
eltRef.setAttribute(“xmi.idref”, resolveID(element.getElement));
tmp.appendChild( eltRef );
ret.appendChild(tmp);
```

CoreSemanticModelBridgeTransformer:

```
Element ret = document.createElement(“UML:CoreSemanticModelBridge”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“presentation”, element.getPresentation());
```

LeafElementTransformer:

```
Element ret = document.createElement(“UML:LeafElement”);
ret.setAttribute(“visible”, “” + element.isVisible());
NodeList fromParent = super.transform(element, doc).getChildNodes();
for(int i = 0; i < fromParent.getLength(); i++)
    ret.appendChild(fromParent.item(i));
```

ImageTransformer:

```
Element ret = document.createElement(“UML:Image”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“visible”, “” + element.isVisible());
ret.setAttribute(“url”, element.getUrl());
ret.setAttribute(“mimeType”, element.getMimeType());

NodeList fromParent = super.transform(element, doc).getChildNodes();
for(int i = 0; i < fromParent.getLength(); i++)
    ret.appendChild(fromParent.item(i));
```

TextElementTransformer:

```
Element ret = document.createElement(“UML:TextElement”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“visible”, “” + element.isVisible());
ret.setAttribute(“text”, element.getText());

NodeList fromParent = super.transform(element, doc).getChildNodes();
for(int i = 0; i < fromParent.getLength(); i++)
    ret.appendChild(fromParent.item(i));
```

DimensionTransformer:

```
Element ret = document.createElement(“UML:Dimension”);
ret.setAttribute("width", element.getWidth());
ret.setAttribute("height", element.getHeight());
```

PointTransformer:

```
Element ret = document.createElement(“UML:Point”);
ret.setAttribute("x", “”+element.getX());
ret.setAttribute("y", “”+element.getY());
```

BeizerPointTransformer:

```
Element ret = document.createElement(“UML:BezierPoint”);
ret.setAttribute("x", “”+element.getX());
ret.setAttribute("y", “”+element.getY());

Element controlPoints = document.createElement(“UML:BezierPoint.controls”);
for(Point p : controls)
    controlPoints.appendChild(diagramTransformer.transform(p, document));
ret.appendChild(controlPoints)
```

GraphicPrimitiveTransformer

```
Element ret = document.createElement(“UML:GraphicPrimitive”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“visible”, “” + element.isVisible());
```

```
NodeList fromParent = super.transform(element, doc).getChildNodes();
for(int i = 0; i < fromParent.getLength(); i++)
    ret.appendChild(fromParent.item(i));
```

PolylineTransformer:

```
Element ret = document.createElement(“UML:Polyline”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“visible”, “” + element.isVisible());
ret.setAttribute(“closed”,  “”+element.isClosed());

Element tmp = document.createElement(“UML:Polyline.waypoints”);
for(Point p : element.getWaypoints())
    tmp.appendChild(diagramTransformer.transform(p, document));
ret.appendChild(tmp);

NodeList fromParent = super.transform(element, doc).getChildNodes();
for(int i = 0; i < fromParent.getLength(); i++)
    ret.appendChild(fromParent.item(i));
```

EllipseTransformer:

```
Element ret = document.createElement(“UML:Ellipse”);
ret.setAttribute(“xmi.id”, resolveID(element));
ret.setAttribute(“visible”, “” + element.isVisible());
ret.setAttribute(“radiusX”,  “”+element.getRadiusX());
ret.setAttribute(“radiusY”,  “”+element.getRadiusX());
ret.setAttribute(“rotation”,  “”+element.getRotation());
ret.setAttribute(“startAngle”,  “”+element.getStartAngle());
ret.setAttribute(“endAngle”,  “”+element.getEndAngle());

Element tmp = document.createElement(“UML:Ellipse.center”);
tmp.appendChild(diagramTransformer.transform(element.getCenter()));
ret.appendChild(tmp);
```