XMI Reader UML Activity Graph Plugin Component 1.0 Specification

1. Design

The XMI Reader UML Activity Graph component is a plugin for XMI Reader component. It implements the ContentHandler (from org.xml.sax) interface and is able to parse all the elements for the classes in the UML Model State Machines and Activity Graphs.

1.1 General approach

1.1.1 Anatomy of the proposed design

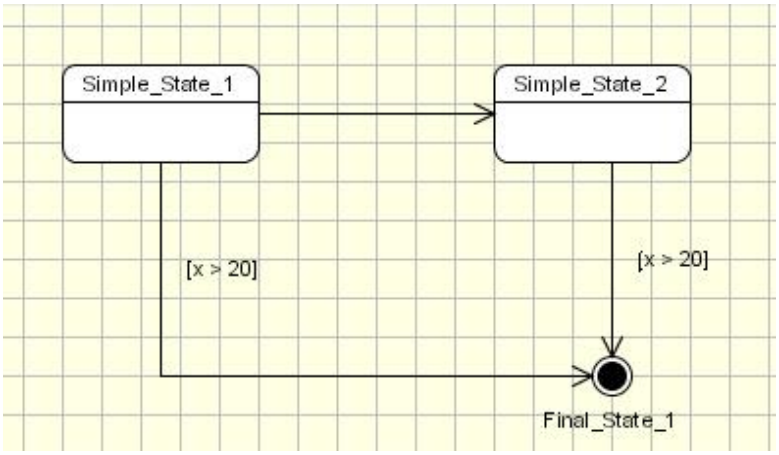
The basic tasks of this design can be broken up as follows:

1. We will need to have the ability to parse out the XMI data that defines two sets of sub object model entities:
    a. State Machine
    b. Activity Graph

    We will mostly deal with parser events such as startElement and endElement, which will hold the bulk of our processing.
2. Actually we will even go further, and create something of a factory, which will generate the appropriate object based on the type of Element that is being read in XMI. Thus we will have the ability to plug-in new classes of model items (such as StateMachine, StateVertex, or ActionState) into our factory.

Lets look at a simple example of a State Machine (StateMachine_1) and the XMI file for it:

Diagram 1. An example of a state machine graph



Here is the xml (xmi) for it in UML 1.5:

Listing 1. Example of state machine xmi (for diagram 1)

```
<UML:StateMachine xmi.id = 'I10ad419m10729d8da08mm7f50' name = 'StateMachine_1'
  isSpecification = 'false'>
  <UML:StateMachine.context>
    <UML:Class xmi.idref = 'I10ad419m10729d8da08mm7f51'/>
  </UML:StateMachine.context>
  <UML:StateMachine.top>
    <UML:CompositeState xmi.id = 'I10ad419m10729d8da08mm7f4f' name = ''
    isSpecification = 'false'
```

```
    isConcurrent = 'false'>
    <UML:CompositeState.subvertex>
      <UML:SimpleState xmi.id = 'I10ad419m10729d8da08mm7f45' name = 'Simple_State_1'
        visibility = 'public' isSpecification = 'false'>
        <UML:StateVertex.outgoing>
          <UML:Transition xmi.idref = 'I10ad419m10729d8da08mm7f21'/>
          <UML:Transition xmi.idref = 'I10ad419m10729d8da08mm7f18'/>
        </UML:StateVertex.outgoing>
      </UML:SimpleState>
      <UML:SimpleState xmi.id = 'I10ad419m10729d8da08mm7f3a' name = 'Simple_State_2'
        visibility = 'public' isSpecification = 'false'>
        <UML:StateVertex.outgoing>
          <UML:Transition xmi.idref = 'I10ad419m10729d8da08mm7f2a'/>
        </UML:StateVertex.outgoing>
        <UML:StateVertex.incoming>
          <UML:Transition xmi.idref = 'I10ad419m10729d8da08mm7f18'/>
        </UML:StateVertex.incoming>
      </UML:SimpleState>
      <UML:FinalState xmi.id = 'I10ad419m10729d8da08mm7f2f' name = 'Final_State_1'
        visibility = 'public' isSpecification = 'false'>
        <UML:StateVertex.incoming>
          <UML:Transition xmi.idref = 'I10ad419m10729d8da08mm7f2a'/>
          <UML:Transition xmi.idref = 'I10ad419m10729d8da08mm7f21'/>
        </UML:StateVertex.incoming>
      </UML:FinalState>
    </UML:CompositeState.subvertex>
  </UML:CompositeState>
</UML:StateMachine.top>
<UML:StateMachine.transitions>
  <UML:Transition xmi.id = 'I10ad419m10729d8da08mm7f2a' isSpecification = 'false'>
    <UML:Transition.guard>
      <UML:Guard xmi.id = 'I10ad419m10729d8da08mm7f05' name = '' visibility = 'public'
        isSpecification = 'false'>
        <UML:Guard.expression>
          <UML:BooleanExpression xmi.id = 'I10ad419m10729d8da08mm7f04' language = 'java'
            body = 'x &gt; 20'/>
        </UML:Guard.expression>
      </UML:Guard>
    </UML:Transition.guard>
    <UML:Transition.source>
      <UML:SimpleState xmi.idref = 'I10ad419m10729d8da08mm7f3a'/>
    </UML:Transition.source>
    <UML:Transition.target>
      <UML:FinalState xmi.idref = 'I10ad419m10729d8da08mm7f2f'/>
    </UML:Transition.target>
  </UML:Transition>
  <UML:Transition xmi.id = 'I10ad419m10729d8da08mm7f21' isSpecification = 'false'>
    <UML:Transition.guard>
      <UML:Guard xmi.id = 'I10ad419m10729d8da08mm7f0f' name = '' visibility = 'public'
        isSpecification = 'false'>
        <UML:Guard.expression>
          <UML:BooleanExpression xmi.id = 'I10ad419m10729d8da08mm7f0e' language = 'java'
            body = 'x &gt; 20'/>
        </UML:Guard.expression>
      </UML:Guard>
    </UML:Transition.guard>
    <UML:Transition.source>
      <UML:SimpleState xmi.idref = 'I10ad419m10729d8da08mm7f45'/>
    </UML:Transition.source>
    <UML:Transition.target>
      <UML:FinalState xmi.idref = 'I10ad419m10729d8da08mm7f2f'/>
    </UML:Transition.target>
  </UML:Transition>
  <UML:Transition xmi.id = 'I10ad419m10729d8da08mm7f18' isSpecification = 'false'>
    <UML:Transition.source>
      <UML:SimpleState xmi.idref = 'I10ad419m10729d8da08mm7f45'/>
    </UML:Transition.source>
    <UML:Transition.target>
      <UML:SimpleState xmi.idref = 'I10ad419m10729d8da08mm7f3a'/>
    </UML:Transition.target>
  </UML:Transition>
```

```
  </UML:StateMachine.transitions>
</UML:StateMachine>
```

The one thing to note is that this is just a simple xml file with a very specific (DTD is provided in \docs\AS.dtd) format. The way that we will have this handled is by simply walking down the DOM of the xmi and by pushing parsing specific elements.
For example we could envision that parsing the above information would produce the following order of events, where each event is depicted as an entry into an xml element:

Example 1. Elements that will be translated into actual objects

```
<UML:StateMachine>                                    // we build a StateMachine instance
  <UML:StateMachine.context>                          // We build and add context
  <UML:StateMachine.top>                              // We build and add a top state
    <UML:CompositeState>                              // which is a composite state
      <UML:CompositeState.subvertex>                  // which is made of 3 simple states
        <UML:SimpleState name = 'Simple_State_1'>     // This is the first state
          <UML:StateVertex.outgoing>                  // which has 2 outgoing transitions
            <UML:Transition>                          // outgoing transition 1a
            <UML:Transition>                          // outgoing transition 2a
        <UML:SimpleState name = 'Simple_State_2'>     // This is the second state
          <UML:StateVertex.outgoing>                  // which has 1 outgoing transition
            <UML:Transition>                          // outgoing transition 1b
          <UML:StateVertex.incoming>                  // which has 1 incoming transition
            <UML:Transition>                          // incoming transition 1b
        <UML:FinalState name = 'Final_State_1'>       // This is the final state
          <UML:StateVertex.incoming>                  // which has 2 incoming transitions
            <UML:Transition>                          // incoming transition 1c
            <UML:Transition>                          // incoming transition 2c
  <UML:StateMachine.transitions>                      // Actual transition details
    <UML:Transition>                                  // We build a transition element
      <UML:Transition.guard>                          // which includes a guard
        <UML:Guard>                                   // we build the guard
          <UML:Guard.expression>                      // which includes and expression
            <UML:BooleanExpression body = 'x&gt;20'/> // we build a boolean expression
      <UML:Transition.source>                         // Source of the transition
        <UML:SimpleState>                             // it starts in Simple_State_1
      <UML:Transition.target>                         // target of the transition
        <UML:FinalState>                              // it ends in Final_State_1
    <UML:Transition>                                  // Another transition element
      <UML:Transition.guard>                          // guard
        <UML:Guard>                                   // we build it
          <UML:Guard.expression>                      // the guard expression
            <UML:BooleanExpression body = 'x&gt;20'/> // we build it
      <UML:Transition.source>                         // it starts in Simple_State_1
      <UML:Transition.target>                         // target of the transition
        <UML:FinalState>                              // it ends in Final_State_1
    <UML:Transition>                                  // Another transition element
      <UML:Transition.source>                         // Source of the transition
        <UML:SimpleState>                             // it starts in Simple_State_1
      <UML:Transition.target>                         // target of the transition
        <UML:SimpleState>                             // it ends in Final_State_2
```

One very important aspect to note is that we are dealing with a pointer based data structure in the sense that some elements are defined using ids, which then get resolved into proper objects. Thus we need to be aware that in XMI we will have references and objects (classes)

Consider how transitions are actually defined. We have an actual physical definition of a transition as in:

Example 2. Transition definition in xmi

```
<UML:Transition xmi.id = 'I10ad419m10729d8da08mm7f2a' isSpecification = 'false'>
  <UML:Transition.guard>
    <UML:Guard xmi.id = 'I10ad419m10729d8da08mm7f05' name = '' visibility = 'public'
      isSpecification = 'false'>
      <UML:Guard.expression>
        <UML:BooleanExpression xmi.id = 'I10ad419m10729d8da08mm7f04' language = 'java'
          body = 'x &gt; 20'/>
      </UML:Guard.expression>
    </UML:Guard>
  </UML:Transition.guard>
  <UML:Transition.source>
    <UML:SimpleState xmi.idref = 'I10ad419m10729d8da08mm7f3a'/>
  </UML:Transition.source>
  <UML:Transition.target>
    <UML:FinalState xmi.idref = 'I10ad419m10729d8da08mm7f2f'/>
  </UML:Transition.target>
</UML:Transition>
```

And then we have a reference to this as in:

Example 3. Transition reference example

```
<UML:StateVertex.incoming>
  <UML:Transition xmi.idref = 'I10ad419m10729d8da08mm7f2a'/>
  <UML:Transition xmi.idref = 'I10ad419m10729d8da08mm7f21'/>
</UML:StateVertex.incoming>
```

This means that our algorithm will have to basically build a mapping of ids to entities and then fill in the complete model with instances of classes that are substituted for the ids.

Note that the approach for an activity diagram will be exactly the same.

1.1.2 Using UML tags to create corresponding value objects

The first thing is to know how to create proper model objects based on parsed elements. This actually rather uncomplicated and works on the premise that a number of tags will map directly to implementation classes. For example the <UML:StateMachine …/> tag maps directly into our StateMachineImpl class, and thus when we encounter this tag we will be most probably creating (unless it is a reference) a new StateMachineImpl.
Here is a table that maps the main classes used here to their xmi tags:

Table 1. State Machine mapping from xmi to Object Model

Xmi element	Object Model element	Description
UML:StateMachine	StateMachineImpl	Class
UML:Transition	TransitionImpl	Class
UML:Guard	GuardImpl	Class
UML:PseudoState	PseudoStateImpl	Class
UML:CompositeState	CompositeStateImpl	Class
UML:SimpleState	SimpleStateImpl	Class
UML:FinalState	FinalStateImpl	Class

Table 2. Activity Graph mapping from xmi to Object Model

Xmi element	Object Model element	Description
UML:ActivityGraph	ActivityGraphImpl	Class
UML:ActionState	ActionStateImpl	Class
UML:CallState	CallStateImpl	Class
UML:ObjectFlowState	ObjectFlowStateImpl	Class

But we have a special case here as well:

1. If the element has a attribute named xmi.id, we first check if a reference object was created for this (due to a forward xmi.idref declaration) and create a new object if it wasn't.
2. If this attribute doesn't exist we then look for an xmi.idref attribute.
    a. If we have the attribute then we use its value to lookup to see if this object was already created.
    b. If it has been created we simply fetch it. And if it has not been created, we create it and place the empty object instance into a table of forward references.

Once we have an object in our hand we can start setting its contents.

1.1.3 Using reflection to map setters to their xmi counterparts

The object instances that we create are empty, and they need to be filled up with the data from xmi.

The most challenging aspect of this task is to parse out enough information to match the data with the proper setter in the class. The idea is that we will use the fact that the DTD definition for the xmi UML is exactly matched in the setters and getters of all the models in the supporting components. This means that when we encounter something like < UML:Transition.stateMachine …/> we can very easily 'guess' the correct setter of the TransitionImpl class (which maps to UML:Transition) will be setStateMachine.

This means that most of the time we will easily be able to construct the correct setter call. But there are special cases that deal with Collections, Lists and boolean variables. We can follow this simple algorithm as follows (assume some elementName):

1. We check if we have a setElementName method in the instance.
2. [If failed] we next check if this was a Collection and we look for an addElementName method

Another case is if the element name is in the format of isElementName.

3. We check if we have a setElementName (we drop the 'is')

These are currently the only three possibilities (based on the DTD) for setters.

1.1.4 How do we know when we have a setter?

1. Anytime we have an attribute we have a setter (except that we do not treat the xmi.id and xmi.idref) as setters.
2. Any time we have a tag in the format of <UML.inner.outer … /> the outer is treated as a setter. So for example if we had <UML:Transition.guard . . .> guard would be the outer element and we would have a setter in the form of setGuard(…)

1.1.5 Which object do we set this setter on?

Since this is a recursive process the object we are setting is always the object that we have in our hand so to speak.

1.1.6 Element ids and reference ids

We also need to make a distinction when we are using `xmi.id` vs. `xmi.idref`. The difference is crucial to parsing. The xmi.id refers to the actual entity (i.e. we will be creating an instance for it) whereas xmi.idref is a reference to such an entity. In effect it is a pointer to the definition of an entity.
Please note that the xmi.id xmi.idref is what the `XMIReader` component uses when its `XMIReader.foundElements` and `XMIReader.forwardReferences` are being used.
`xmi.id` is used as a key in `foundElements` and `xmi.idref` is usually used in `forwardReferences`.

1.2 Design Patterns

Also **Factory** pattern is used to create instances of different model classes based on some key.

1.3 Industry Standards

UML 1.5
XMI 1.2

1.4 Required Algorithms

The main algorithm is of course the actual parsing of the document for the specific elements.

1.4.1 Parsing the document's elements

The implemented SAX handler is really doing things through a call back. The main methods that we care about are `startElement` and `endElement`, which tell us that we have entered or exited an element.
The general aspect will be as follows:

startElement:
For each element that we use reflection as stated in section 1.1.2 and 1.1.3:
Lets say that we obtained an element with name `"UML:StateMachine"`
Based on the element name this would have happened:

```
// We create a new StateMachine instance (stateMachine)
Object temp =  getModelElementFactory.createModel("UML:StateMachine");

// If the element is UML:ActivityGraph definition, then add the ActivityGraph
// instance to the uml model manager
this.modelManager.addActivityGraph((ActivityGraph) obj);

// We set all the provided data in properties according to reflection as
// described in 1.1.3
// We also get the xmi.id and we use this id as follows:
getXMIReader().putElement(xmi.id, stateMachine);
// we also set the last object to be StateMachine
setLastObject(StateMachine);

// save the current element name and its associated uml model element instance in
// stack
```

Note that similar ideas will apply to all the UML elements. The developer should have no issues expanding the above simple exposition.

endElement:

```
// If the xmi element represents a uml model element instead of a property
// update the parent property element for the parent uml model element
// for example
// <UML:StateMachine.top>
//     <UML:CompositeState .../>
// </UML:StateMachine.top>
// The current element is UML:CompositeState, and the parent element is
// UML:StateMachine.top, update the top property for StateMachine instance

// pop the element name and its associated uml model element from stack
```

1.5 Component Class Overview

1.5.1 com.topcoder.xmi.reader.handlers.uml.activitygraph

ActivityGraphXMIHandler <<concrete class>>
This is a concrete (but indirect) implementation of the `ContentHandler` interface. This is done by extending the `DefaultXMIHandler` abstract class, which implements some convenience methods that deal with XMIReader. This is important since we need to utilize XMIReader instance in the actual implementation.
This class is responsible for parsing out State Machine as well as Activity Graph model elements. Note that since Activity Graph is extended from State Machine this is a natural fit and there is no need to split the handler into two separate entities. This handler uses the ModelElementFactory to instantiate the proper model class instance. Basically as we parse the xml elements we use the element designation name (like `UML:ActivityGraph` for example) as a key to create an instance of a class in the Object Model which represents the `UML:ActivityGraph` which in our case (depending on configuration of course) would be `ActivityGraph`.

As required this implementation is thread-safe. This handler acts mostly like a utility. The only time it relies on state information is when it interacts with XMIReader (thread-safe), ModelElementFactory (thread-safe) and UMLModelManager (assumed to be thread-safe)

ModelElementFactory <<concrete class>>
This is a configurable factory, which creates UML Model class instances based on a mapping from xml element name to a class name of the model element that currently represents it.
This implementation is thread-safe. Even though it is not anticipated that many thread will be hitting the handler, it was decided that it might be worthwhile to have a single copy of the factory (to save memory and configuration read time) Synchronization should be done in a block to minimize performance impact.

1.6 Component Exception Definitions

1.6.1 Custom Exceptions

Only two simple creation exceptions has been created

ElementCreationException:

Thrown when the ModelElementFactory cannot create the requested instance. This could be due to reflection issues, sandbox security issues, or something else.

ConfigurationException:
It will be thrown by the two ModelElementFactory constructors if there are issues with configuration (i.e. wrong format or missing variables) or if Configuration Manager thrown ConfigManagerException .

1.6.2 System and general exceptions

In general this component will check for empty string input or null reference parameter input. It will also check of arrays have any null references or empty strings.

IllegalArgumentException:
This exception is used for invalid arguments. Invalid arguments in this design are used on some occasions with null input elements. The exact details are documented for each method in its documentation.

SAXException:
This exception is used to signal to the invoking SAX parser that the handler is unable to continue parsing the given data.

1.7 Thread Safety

Implementation is thread-safe since the contract of `XMIHandler` from which we ultimately get the handler for this component, specifies that implementation MUST be thread-safe.

2. Environment Requirements

2.1 Environment

- Development language: Java1.5
- Compile target: Java1.5

2.2 TopCoder Software Components

- **UML Model - Activity Graphs 1.0**
  Used to model activity graphs
- **UML Model - State Machines 1.0**
  Used to model state machines
- **Config Manager 2.1.5**
  Used to implement to read configuration detail for mapping xml element names (q names) with class names, which will be used (via reflection) to create model elements for the xml element.
- **XMI Reader 1.0**
  This is the base component for which we are writing this plugin. We not only will plug-in the handler but we also use the XMIReader itself in this component to manager ids and reference ids for xml elements.
- **UML Model Manager 1.0**
  This is used to store the parsed out and created Activity Graph diagram
- **Base Exception 1.0**
  Used as a parent class for all checked custom exceptions in this design to facilitate controller exception chaining.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.xmi.reader.handlers.uml.activitygraph

3.2 Configuration Parameters

3.2.1 ModelElementFactory configuration for Config manager

Parameter	Description	Values
xml_to_element_mapping	These are 1 or more xml element name mapping to a class name used to create an object, which can store information about the xml element in the UML Model. These are comma delimited values, where the first element is the xml element name and the second element is a class name". ***Required***	"UML:ActivityGraph , com.topcoder.uml.model.activitygraphs.ActivityGraphImpl"

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 support the current State Machine and Activity Graph models the user should configure the mapping as provided in tables 1 and 2 in section 1.1.2:

4.3 Demo

The demonstration integrates this component and the XMIReader component for parsing xmi file, and the usage of ModelElementFactory is shown as well.

The configuration for XMIReader is important for the demo, for detail configuration information, please consult the file located in "test_files/reader_config.xml".

Set up the XMIReader instance and ActivityGraphXMIHandler instance for parsing:

```
TestHelper.loadSingleXMLConfig(XMIReader.class.getName(),
    "test_files" + File.separator + "reader_config.xml");
TestHelper.loadSingleXMLConfig(ModelElementFactory.class.getName(),
    "test_files" + File.separator + "ModelElementFactoryConfig.xml");
reader = new XMIReader();

ActivityGraphXMIHandler handler = (ActivityGraphXMIHandler)
    reader.getHandler("UML:StateMachine");
handler.setModelElementFactory(
    new ModelElementFactory(ModelElementFactory.class.getName()));
```

4.3.1 It demonstrates the usage of XMIReader and ActivityGraphXMIHandler to parse a xmi file which has State Machine diagram.

```
reader.parse(new File("test_files" + File.separator + "statemachine_sample.xmi"));

// Get the StateMachine instance via xmi id
StateMachine stateMachine = (StateMachine)
    reader.getElement("I10ad419m10729d8da08mm7f50");

// Get the related uml elements from the StateMachine
State state = stateMachine.getTop();
 Collection<Transition> transitions = stateMachine.getTransitions();
```

4.3.2 It demonstrates the usage of XMIReader and ActivityGraphXMIHandler to parse a xmi file which has Activity Graph diagram.

```
reader.parse(new File("test_files" + File.separator + "activitygraph_sample.xmi"));

// Get the ActivityGraph instance via xmi id
ActivityGraph activityGraph = (ActivityGraph) reader.getElement(
    "-64--88-1-88--4d34a780: 10f7b30207c:-8000:00000000000007EF");

// Get the related uml elements from the ActivityGraph
State state = activityGraph.getTop();
 Collection<Transition> transitions = activityGraph.getTransitions();
```

4.3.3 It demonstrates the functionality of ModelElementFactory.

```
// Create a default instance
ModelElementFactory modelElementFactory = new ModelElementFactory();

// Create an instance with configuration data
modelElementFactory = new ModelElementFactory(ModelElementFactory.class.getName());

// Adds a new mapping to the class
modelElementFactory.addMapping("UML:StateMachine",
    "com.topcoder.uml.model.statemachines.StateMachineImpl");

// Creates an actual instance maps to the specific xml element
modelElementFactory.createModelElement("UML:StateMachine");

// Gets the class name for the given xml element
modelElementFactory.getMapping("UML:StateMachine");

// Return the complete mapping
modelElementFactory.getAllMappings();

// Remove the xml element mapping
modelElementFactory.removeMapping("UML:StateMachine");
```

5. Future Enhancements

- None at this time