XMI Parser 1.0 Component Specification

1. Design

An XMI file contains several UML diagrams - Use Case, Class, Sequence, Collaboration, State, and Deployment and elements like Association, Actor, and Use Case etc. Each diagram (e.g. with the tag UML:Diagram) is a node which contains references to the elements (e.g. UML:UseCase). These elements are nodes themselves which in turn contain references to their diagrams.

The XMI Parser Component provides the ability to determine the UML elements that make up an XMI format of a Poseidon UML model.

The parser can accept a File, Stream or a String as a source of XMI. It uses callback methods on handlers to de-couple parsing and processing. The handlers can be configured with the help of TopCoder Configuration Manager Component. The default namespace contains the configuration for the default handling provided with the component. The component makes it possible to plug-in specific custom handlers for each node or handlers for a set of nodes by specifying configuration properties in a different namespace (as standard Java properties) and passing this namespace to the constructor. In the constructor, the parser registers the handlers for the nodes from the configuration in an internal map. The parser creates only one instance of a particular handler.

The handlers can also be managed programmatically.

The default handler handles set of required nodes (UML diagrams/elements). It provides the following functionality –

1. Builds the collection of diagrams as data objects. Each diagram in turn contains objects for the elements. For example, there is a collection of Class diagrams. Each Class diagram contains a set of Classes.
2. Provides API to access the diagrams of a certain type.

The initial version of the API would contain

a. Method giving a list of diagrams as data objects of a given type;

b. Method giving a list of diagram names of a given type;

c. Method that gives a list of concrete use cases in a given Use Case diagram;

d. Method that gives a list of abstract use cases in a given Use Case diagram;

e. Method to extract information about use case diagram actors;

f. Method to extract information about activity diagram: set of initial states, set of final states, set of transitions for each state etc.

The configuration property names match the UML diagram/element type names in XMI. The parser executes methods on the handler associated with node name. So, handling depends on the presence/absence of the properties in configuration. This would make processing of XMI extensible to include new elements/diagrams. For new element/diagram types or for new handlers, it would be only necessary to add configuration properties and handlers and there would be no need to modify parts of the parser component.

For example, in the ‘default.parser’ namespace, handlers.properties defines,

UseCase=DefaultHandler

Association=DefaultHandler

UseCaseDiagram=DefaultHandler

If specific handlers are required, a different namespace for e.g. ‘specific.parser’ namespace contains,

UseCase=UseCaseHandler

Association=AssocHandler

UseCaseDiagram=UseCaseDiagramHandler

The namespace is set in the constructor of the parser. So, the namespace represents a set of handlers to be involved in processing.

The Parser would first get all the nodes with the name ‘UML:Diagram’(Java API provides convenient way of doing this). This is because, considering the structure of XMI, there will be less overhead, in later linking the other elements to the existing diagrams than the other way round. After processing them(described in detail in the next paragraph), it gets the node UML:Namespace of UML:Model which contains the different element types as child nodes. For each node, it then checks for a responsible handler(details later) in the handlerRegister. The diagram/element type name will be the key in register, with the value being the qualified class name of the handler. If it cannot find an entry, it ignores.

When XMIParser encounters a UML:Diagram node, it gets the <UML:GraphElement.semanticModel> that is ***directly*** under this Diagram node. It then gets the <UML:SimpleSemanticModelElement> under this node and gets the value of ‘typeInfo’ attribute. This value tells what the diagram type is. This is also matches the constant string in UMLDiagramTypes class. The parser then looks for the key in handlerRegister with the same name and gets the handler name if present. It looks if this handler is already available in the handlerMap. If yes, it uses it , else it instantiates the handler. It creates an entry in the handlerMap where the key is the qualified handler name and the value is the handler object. From then onwards, whenever the parser encounters that particular diagram type, it gets the handler from the map and executes the handleNode passing the current <UML:Diagram> node and the type of diagram (typeInfo attribute) to it.

When the parser encounters an element node, which is the child nodes in the <UML:Namespace> of <UML:Model> node, it gets the node name which also matches the constants in UMLElementTypes class. It follows the same procedure of acquiring the handler object, adding to handlerMap etc. It then invokes the handleNode with the node and the element type.

The parser should also handle <UML:ActivityGraph> element and its registered sub-elements to collect information on activity diagram states and transitions. Handling is performed in the similar way as for <UML:Namespace> and <UML:Model>.

The parser also has a method getHandler (<node type>) for clients to access such handlers and do any additional operations. The clients can access this handler from the parser (e.g. getHandler (‘Diagram’)) and retrieve additional information through the API provided by the handler.

Default Handler:

This handler needs to maintain state because methods for each diagram/element are invoked by the parser at different times. The handler builds a data structure to store information regarding the diagrams in the model. The API uses this data structure to return information regarding the diagrams/elements.

The handleNode checks the type of element/diagram passed to it which is the second argument. It then invokes the method create<element/diagramType> using reflection. In the default implementation, if the node name is 'UseCaseDiagram', it calls the createUseCaseDiagram() method. If the node name is 'UseCase', it calls the createUseCase(). Extensions or new implementations can follow the same process for handling additional types.

In createUseCaseDiagram(), it creates a UMLUseCaseDiagram data object and fills the information like diagram name, id, type and description. The type will be the same as the current node type which also matches a constant value in UMLDiagramTypes class. Then it gets all the <UML:UseCase> child nodes. For each, UseCase node, it adds an entry in the elementsMap of UMLUseCaseDiagram, where the key is the id the node and the value is null. The UMLUseCaseDiagram object is added to the 'diagrams' list.

In createUseCase() method, get the id of the node passed(which the node corresponding to the UseCase). Build the UMLUseCase object and fill the details. Get the diagrams from 'diagrams' list which are of type UMLUseCaseDiagram. From the elementsMap of this diagram object, match the key to the id of the element node. Add this UMLUseCase object to the elementsMap with that key. This way by end of processing the elementsMap will contain all UseCase elements. Because the XMIParser first processes all diagrams before the elements, there should be no scope of elements not finding a suitable diagram.

Parsing of activity diagram information and information on use case diagram actors requires several additional private methods in the default handler: createActor, createActivityDiagram, createStateMachine_top, createStateMachine_transitions. Implementation of this methods are similar to the previously described one, please refer to method docs for more details.

Data Objects:

For the current requirements, the following data objects will be supported by the default handler.

1. UMLDiagram (Superclass of all diagrams)
2. UMLClassDiagram
3. UMLUseCaseDiagram
4. UMLStateDiagram
5. UMLActivityDiagram
6. UMLCollaborationDiagram
7. UMLSequenceDiagram
8. UMLComponentDiagram
9. UMLElement (Superclass of all elements)
10. UMLUseCase (* composed by UMLUseCaseDiagram)
11. UMLActor

12. UMLState
13. UMLTransition

Each object contains attributes and operations specific to that type of element. For e.g., UMLClass may contain a collection of its attributes, methods and accessors for these.

Exception Handling:

When a handler is declared in the configuration for a particular UML diagram/element node the handleNode is invoked which in turn tries to call the create<diagram/elementType> for e.g createUseCaseDiagram, createUseCase. If such a method is not defined or if there is any other problem as described in the java.lang, java.lang.reflect API, certain exceptions are thrown (like NoSuchMethodException, IllegalArgumentException). The handleNode wraps such exceptions in XMIHandlerException and rethrows which is handled in XMIParser. The XMIParser in turn wraps it in the XMIParserException and rethrows it.

Thus the component realizes chain-exception paradigm.

1.1 Design Patterns

Strategy Pattern: Separating the handling mechanism from the parser is important to plugin new handlers. Each handler represents a new behavior encapsulated in its handleNode() method separate from the parser. The DefaultXMIHandler is an example of Strategy.

Façade Pattern: The DefaultXMIHandler provides interface methods which makes it easier for the client to obtain information like the concrete use cases in a given use case diagram.

1.2 Industry Standards

UML, XMI

1.3 Required Algorithms

The application flow with an example is given below. Referring to the class diagram, sequence diagrams will be helpful-

Example: The model has a Use Case Diagram. This diagram has an Actor, a Use Case and an association between them

1. Create configuration file handler.properties in the namespace “com.topcoder.util.xmi.parser.default.handler”. This contains the following properties – Diagram=com.topcoder.util.xmi.parser.handler.DefaultXMIHandler

UseCase=com.topcoder.util.xmi.parser.handler.DefaultXMIHandler

We do not declare a handler for the Actor and Association

2. Instantiate XMIParser with the namespace above.
3. In the constructor, first load the configuration using the configuration manager and fill the handlerRegister map. Now it has the following key-value pairs.
Diagram: com.topcoder.util.xmi.parser.handler.DefaultXMIHandler,
UseCase: com.topcoder.util.xmi.parser.handler.DefaultXMIHandler
4. Call parser.parse(<file to be parsed>)
5. Create a XML DOM out of the xmi input file using the parser API provided with SUN JDK
6. Get the <UML:Diagram> nodes using Element.getElementsByTagName as list
7. Get the next <UML:Diagram> node.
8. Get the <UML:GraphElement.semanticModel> node that is the direct child of this node. Get the <UML:SimpleSemanticModelElement> under this and get the value of "typeInfo" attribute. This value tells what the diagram type is. It also matches the string constant in UMLDiagramTypes class.
9. If this attribute value is present in the handlerRegister as key, get the name of handler. Else ignore and go to (7).
10. If typeInfo attribute value is 'UseCaseDiagram', you will get 'com.topcoder.util.xmi.parser.handler.DefaultXMIHandler'
11. From handlerMap(notice that this is different from handlerRegister), get the value of 'com.topcoder.util.xmi.parser.handler.DefaultXMIHandler' which is the key obtained from handlerRegister.
12. If the key is present, get the value which is the handler object. go to (15)
13. If not present, create an instance of 'DefaultXMIHandler' using newInstance().
14. In the map handlerMap create an entry with key - 'com.topcoder.util.xmi.parser.handler.DefaultXMIHandler' and value – defHandler (which is instance of XMIHandler created)
15. Call the method handleNode() on handler interface with the current node and the diagram type as arguments(<UML:Diagram node>, "UseCaseDiagram").
16. In handleNode of DefaultXMIHandler, based on the node type which is the argument, in this case, 'UseCaseDiagram', call the method (which is defined in the same class) create<type> reflectively, here, createUseCaseDiagram with the node and type as arguments.

    In createUseCaseDiagram method, create a UMLUseCaseDiagram data object and fills the information like diagram name, id, type and description.

    Set the attributes this way -

    diagramName – 'name' attribute of the node

    diagramId – 'xmi.id' attribute of the node

    diagramType – type passed as parameter. This also matches a constant in UMLDiagramTypes class

    ```
    <UML:Diagram xmi.id="I15155e1m10206505306mm7e75"
    isVisible="true" name="UseCase diagram_1" zoom="1.0">
    ```

17. Then get all the <UML:UseCase> child nodes. For each, UseCase node, add an entry in the elementsMap of UMLUseCaseDiagram, where the key is the id in UseCase node. Add the diagram object to the diagrams list.Go to (7)
18. Now in the parser get the <UML:Namespace> node in the <UML:Model> node. This contains all the elements in all diagrams. In this case, the node names reflect the element names like <UML:UseCase> represent a use case element. We will refer to this list of nodes as element nodes.
19. Get the next node from the element node list. Get the node name. Find if this name is a key in handlerRegister. Get the handler name. If this handler name is present as key in handlerMap, get the value which is the handler. Else create the handler object and add the key-value to the handlerMap. In this case, the only nodes processed will will be <UML:UseCase> nodes and the DefaultXMIHandler is already present in the map. So, we simply get the object.
20. Invoke the handleNode() method on handler with the element node and the node type as parameters.
21. In the handleNode functions as described in (16). In this case, it invokes createUseCase passing the node and type to it. Get the id of the node passed(which the node corresponding to the UseCase). Build the UMLUseCase object and fill the details. Get the diagrams from 'diagrams' list which are of type UMLUseCaseDiagram. From the elementsMap of this diagram object, match the key to the id of the element node. Add this UMLUseCase object to the elementsMap with that key. This way by end of processing the elementsMap will contain all UseCase elements. Because the XMIParser first processes all diagrams before the elements, there should be no scope of elements not finding a suitable diagram.
22. Go to (19)

// End of process description

For the other Diagram/element types, follow the same procedure for enhancing the handler. The parser need not be changed anymore. Only the configuration and the handler(s). Use the other sub-classes of UMLDiagram/UMLElement as shown in the class diagram. For this requirement, we do not need to build element maps for other type of diagrams.

Methods:

For the method to get diagrams of a given type, get the handler from XMIParser. Get the diagrams list. Return the instances of type UML<diagram type>. The parameter must be one of the constants defined by UMLDiagramTypes class. If no diagrams are found, or an invalid type is supplied, null is returned.

For the method to get abstract use cases from a given use case diagram name, get the diagram from the diagrams list whose name matches the parameter. Check if the type is UMLUseCaseDiagram. Call getUseCases(). This method returns the elements from the elementsMap which are instances of UMLUseCase. Return list of the subset of the elements, whose isAbstract attribute is true.

1.4 Component Class Overview

IXMIParser:
The interface implemented by the component's XMI Parser.

XMIParser:
\This is the main parser in the component. It uses DOM parsing and can be configured with the following-

1. namespace - Namespace in the Configuration Manager which allows the parser to retrieve the class names of the handlers for node encountered

XMIHandler:
All handlers must implement this interface which defines the handleNode () method. The XMIParser uses this interface to call this method on the handlers

DefaultXMIHandler:
Provides default implementation of the handler. Also contains several API methods to access the diagram information. Implements the handleNode () which, based on the type calls create<type> methods defined in itself reflectively.
The handler builds a data structure to store information regarding the diagrams/elements in the model. The API uses this data structure to return information regarding the diagrams/elements.

UMLDiagram:
Data object representing the superclass of all diagrams.

UMLClassDiagram:
Data object to hold information about UML Class diagram. Inherits UMLDiagram

UMLUseCaseDiagram:
Data object to hold information about UML Use Case diagram. Inherits UMLDiagram

UMLStateDiagram:
Data object to hold information about UML State diagram. Inherits UMLDiagram

UMLActivityDiagram:
Data object to hold information about UML Activity diagram. Inherits UMLDiagram

UMLCollaborationDiagram:
Data object to hold information about UML Collaboration diagram. Inherits UMLDiagram

UMLSequenceDiagram:
Data object to hold information about UML Sequence diagram. Inherits UMLDiagram

UMLDeploymentDiagramt:
Data object to hold information about UML Deployment diagram. Inherits UMLDiagram

UMLElement:
Data object superclass for UML elements

UMLUseCase:
Data object for a use case. Inherits UMLElement

The UML<type> classes are used by the DefaultHandler to hold information regarding the elements/diagrams in the model.

UMLDiagramTypes:
Contains the constants representing diagram types.

UMLElementTypes:
Contains the constants representing element types.

UMLState:
This class represents a single UML activity graph state. Each state is described by the following information: an XMI id, a state type, a state name.

UMLTransition:
This class represents a single UML activity graph transition. The transition is described by the following data: XMI id of the transition, XMI id of the from node, XMI id of the to node, transition name, guard condition for transition to be executed, guard effect for the executed transition.

UMLActivityDiagram:
This class represents a UML activity diagram. Note that this class keeps no additional information. All necessary information is extracted from the map of UMLElements maintained by the base class.

UMLStateType:
Represents a string description of the UML state type. It is assigned in the constructor and not changed afterwards. The value can be retrieved using public getType() method. It can never be null or an empty string.

1.5 Component Exception Definitions

[List the custom Exceptions in the component design and explain the usage of each and scenarios when they are thrown. Use the format below.]

ParserException

Exception thrown by <code>Parser</code> interface, it wraps any Exception caught by the parser when parsing a given input.

XMIHandlerException:

Exception thrown in handlers during any parsing exception.

The handleNode () method in XMIHandler throws this exception. While finding the method create<nodeType>, if Class throws a NoSuchMethodException, it is wrapped in XMIHandlerException and re-thrown.

The reflection exceptions that might occur in this method are also wrapped in this exception and rethrown with the message containing “Could not invoke method create<nodeType>”

XMIParserException:

Exception thrown in parser.

The parser wraps any exception(including the XMIHandlerException) that happens during processing and rethrows as this exception. The original exception is preserved as the cause.

1.6 Thread Safety

The parser acts on one given file at a time and holds cache of the single instances of handler. The handlers themselves may need to maintain state. As a result, it is not safe for multiple threads to operate on the parser. The client application using the parser should consider the necessary means of sequential parsing.

2. Environment Requirements

2.1 Environment

- Java1.4 is required for compilation and executing test cases.

2.2 TopCoder Software Components

ConfigurationManager

2.3 Third Party Components

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.xmi.parser

3.2 Configuration Parameters

The configuration manager is used by the parser to retrieve properties from ‘handler.properties’ in a given namespace. The namespace defaults to ‘com.topcoder.util.xmi.parser.default.handler’. The config file format will be CONFIG_PROPERTIES_FORMAT

The value is a fully qualified class name of handler

For the requirements of the default handler, the following configuration suffices

Parameter	Description	Values
UseCaseDiagram	Handler for UseCaseDiagram type	com.topcoder.util.xmi.parser.handler.DefaultHandler
UseCase	Handler for UseCase type	com.topcoder.util.xmi.parser.handler.DefaultHandler

For different handlers –

Parameter	Description	Values
UseCaseDiagram	Handler for UseCaseDiagram type	<handler name>
ClassDiagram	Handler for Class Diagram type	<handler name>
UseCase	Handler for UseCase type	<handler name>
Class	Handler for Class type	<handler name>
Activity	Handler for Activity type	<handler name>
Collaboration	Handler for Collaboration type	<handler name>
Sequence	Handler for Collaboration type	<handler name>
State	Handler for State type	<handler name>
…		

AddAdditional parameters may be defined based on requirements.

3.3 Dependencies Configuration

The XMI Parser has a dependency on Configuration Manager Component

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

The Configuration Manager component must be configured and accessible to this component.

4.3 Demo

Parts of “test.xmi” are included –

```
...
...
<UML:Model xmi.id="sm$663731:10213f2b750:-7ffe" name="model 1" isSpecification="false"
      isRoot="false" isLeaf="false" isAbstract="false">
<UML:Namespace.ownedElement>
 <UML:UseCase xmi.id="sm$663731:10213f2b750:-7ffc" name="Register handlers from configuration"
      visibility="public" isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false" />
 <UML:UseCase xmi.id="sm$663731:10213f2b750:-7ff4" name="Parse for UML diagrams and delegate
      to handlers" visibility="public" isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false"
      />
...
<UML:Namespace.ownedElement>
...
...
<UML:Diagram xmi.id="di$663731:10213f2b750:-7fe0" isVisible="true"
      name="XMI_Parser_Use_Case_Diagram" zoom="1.0">
      ...
      <UML:GraphElement.semanticModel>
            <UML:Uml1SemanticModelBridge xmi.id="di$663731:10213f2b750:-7fd5" presentation="">
                  <UML:Uml1SemanticModelBridge.element>
                        <UML:UseCase xmi.idref="sm$663731:10213f2b750:-7ffc" />
                  </UML:Uml1SemanticModelBridge.element>
            </UML:Uml1SemanticModelBridge>
    </UML:GraphElement.semanticModel>
     ...
     <UML:GraphElement.semanticModel>
            <UML:Uml1SemanticModelBridge xmi.id="di$663731:10213f2b750:-7fd5" presentation="">
                  <UML:Uml1SemanticModelBridge.element>
                        <UML:UseCase xmi.idref="sm$663731:10213f2b750:-7ffc" />
                  </UML:Uml1SemanticModelBridge.element>
            </UML:Uml1SemanticModelBridge>
    </UML:GraphElement.semanticModel>
     ...
</UML:Diagram>
...
...
```

```
Class ParserTest {
      File xmiFile = new File(“test.xmi”);
      String ns = “com.topcoder.util.xmi.parser.test”;
      XMIParser parser = new XMIParser(ns);
      parser.parse(xmiFile);

      MyHandler handler = (MyHandler)parser.getHandler();
```

```
// Get diagram names
List diagramNames =
handler.getDiagramNamesOfType(UMLDiagramTypes.UML_USE_CASE_DIAG
RAM);
System.out.println(diagramNames.toArray());



/*
```

Output:

XMI_Parser_Use_Case_Diagram

```
*/
// Get diagrams of a specific type
List diagrams =
handler.getDiagramsOfType(UMLDiagramTypes.UML_USE_CASE_DIAGRAM);


for(int i=0;i<diagrams.size();i++) {
      UMLUseCaseDiagram ucDiag = (UMLUseCaseDiagram) diagrams.get(i);
       System.out.println(ucDiag.getDiagramName());
}
/*
```

Output:

XMI_Parser_Use_Case_Diagram

```
*/
// Get concrete use cases
List concreteUCs = handler.getConcreteUseCases("MyUseCaseDiagram");
for(int i=0;i< concreteUCs.size();i++) {
      UMLUseCase uCase = (UMLUseCase) concreteUCs.get(i);
       System.out.println(uCase.getElementName());
}
/*
```

Output:

Register handlers from configuration

Parse for UML diagrams and delegate to handlers

```
*/
}
```

5. Future Enhancements

Provide more fine-grained details about the contents of XMI

2. Provide more details about diagrams like associations, cardinality