Using XMI Documenter to build an XMI documentation editor UI

1. Introduction

This article will show how the XMI Documenter component can be used to build an XMI documentation editor which can be used as a replacement for the somewhat sub-optimal documentation editing features of Poseidon.

As the functionality and concepts of XMI Documenter component itself are already explained in the XMI Documenter CS, this document will not explain these details again, but will focus on three main topics:

- How to use the XMI Documenter API to build an editor;
- XMI documenter Implementation details;
- How to setup and run the XMI Documenter UI;

Just to clarify from the start of this article: the article includes a fully functional XMI documentation editor example application that can be used to edit the documentation elements of an XMI model. In case you are not really interested in reviewing all of the details you can jump directly to Chapter 4 and configure and run the already existent UI.

2. How to use the XMI Documenter API

This chapter will describe the main parts of the XMI Documenter API and show how they work together and are used in the demo provided with this article.

The starting point for all XMI documentation editing is the class `XMIDocumentFactory`. This class provides two factory methods (both named `createDocument`, one takes a `java.io.InputStream`, the other one an `org.w3c.dom.Document`) for parsing an XMI input into an `XMIDocument` instance (see `XMIDocumenterUIDemo#load()`):

```
FileInputStream xmiStream = new FileInputStream(selectedFile);
 try {
    // load the XMI document
    currentDocument = factory.createDocument(xmiStream);
    // update the tree
    treeModel.setRoot(new XMIDocumentNode(currentDocument));
    treeModel.reload();
    saveButton.setEnabled(true);
} finally {
    xmiStream.close();
}
```

An `XMIDocument` represents the structure of the parsed XMI file. The instance itself, and all of its members, does not contain all the information that was in the XMI file – it doesn't even contain enough information to reconstruct a valid XMI file from an `XMIDocument` instance. Instead the `XMIDocument` and all of its `ModelElements` are backed by a DOM tree of the parsed XMI file. The read and write of documentation for elements actually occurs by accessing the backing DOM nodes of the corresponding `ModelElements`.

An `XMIDocument` instance is mainly a holder for all classes and interfaces defined in the XMI document. These classes and interfaces can be retrieved using the `getClasses` method. Subsets of all defined classes of the document can be retrieved using the `getClasesForPackage` or `getClassesWithOperations` methods. For details on the handling of interfaces see Chapter 3 of this article. An example of the retrieval of classes from an `XMIDocument` instance can be seen in

`XMIDocumenterUIDemo.XMIDocumentNode#XMIDocumentNode()` – this method creates child nodes for all packages and classes defined in the `XMIDocument` instance:

```
// get classes defined in document
final ClassElement[] ownedElements = document.getClasses();
for (int i = 0; i < ownedElements.length; i++) {
    final ClassElement classElement = ownedElements[i];
    final String packageName = classElement.getPackageName();
  // find or create the package node containing the class node
    final PackageNode packageNode = findPackageNode(packageName);
    // create the class node and add it to the parent package node
    packageNode.addChildNode(new ModelElementTreeNode(classElement,
          packageNode));
}
```

A `ClassElement` contains `AttributeElements` for all fields defined in the class and `OperationElements` for all methods defined in the class. `OperationElements` consist of `ParameterElements`, an optional `ReturnValueElement` and `ExceptionElements`. The whole documentation tree of an XMI document is described by these Elements. The UI editor itself can even abstract the differences between all of these instances and treat all of the instances as `ModelElement` instances. This way all of the instances can be by handled in a generic way - see `XMIDocumenterUIDemo.ModelElementTreeNode` which abstracts the editing and handling of `ModelElements`, e.g. in the constructor:

```
ModelElementTreeNode(final ModelElement element, final TreeNode parent) {
     super(parent);
     this.element = element;
     final ModelElement[] ownedElements = element.getOwnedElements();
     for (int i = 0; i < ownedElements.length; i++) {
         final ModelElement ownedElement = ownedElements[i];
         getChildren().add(new ModelElementTreeNode(ownedElement, this));
     }
}
```

All of the modification logic inside `XMIDocumenterUIDemo` also relies on the API declared by `ModelElement.` Basically the methods `getName` `getDocumentationText` and `setDocumentationText` are used to display and edit all of the elements and their documentation.

Upon selection of a tree node that represents a `ModelElement`, any previously started editing of a `ModelElement` instance is ended by writing the current editor value back to the `ModelEelment` instance - see `XMIDocumenterUIDemo#endElementEditing()`:

```
if (currentlyEditedModelElement != null) {
    final String text = editingArea.getText();
    if (!text.equals(currentlyEditedModelElement.getDocumentationText())) {
        currentlyEditedModelElement.setDocumentationText(text);
    }
    currentlyEditedModelElement = null;
    editingArea.setText("");
    editingArea.setEnabled(false);
}
```

Afterwards the `ModelElement` represented by the selected node becomes the currently edited element and its current documentation text is written to the documentation editor pane - see `XMIDocumenterUIDemo#editModelEelement():`

```
private void editModelElement(final ModelElement modelElement,
                              final ModelElementTreeNode modelElementTreeNode) {
    endElementEditing();
    currentlyEditedTreeNode = modelElementTreeNode;
    currentlyEditedModelElement = modelElement;
    editingArea.setText(currentlyEditedModelElement.getDocumentationText());
    editingArea.setEnabled(true);
}
```

The only point at which this generic handling of `ModelElements` is broken up is the `XMIDocumenterUIDemo.XMIDocumenterTreeRenderer`, that checks the type of all of the `ModelElement` instances using `instanceof` to be able to display the type of the `ModelElement` represented by a TreeNode.

When editing of the XMI document has been finished, the document instance can be written into an `java.io.OutputStream` using `XMIDocument#writeTo()` – see `XMIDocumenterUIDemo#save()` for an example on this:

```
final FileOutputStream out = new FileOutputStream(selectedFile);
try {
    currentDocument.writeTo(out);
    treeModel.setRoot(new XMIDocumentNode(currentDocument));
    treeModel.reload();
} finally {
    out.close();
}
```

After describing all of the core elements of the XMI Documenter API, this article has hopefully helped illustrate how to load, modify and save XMI files using XMI Documenter component.

3. Implementation details of XMI documenter that are worth noting

This chapter describes some of the XMI Documenter implementation details that are not suitable to be mentioned in the API documentation or CS of a component.

The most interesting aspect of the XMI Documenter for designers who want to use this editor might be what is necessary to exist in the XMI document before staring editing of documentation. As the XMI Documenter does not provide any functionality for the creation of model elements, all elements that need to be documented must exist before loading the document. This basically means that all classes have to be declared, all methods and fields to be documented must be created inside Poseidon and all parameters need to be modeled. Special care must be taken of any exception documentation. In the XMI document created by Poseidon, there is no real structural model element that represents a throws declaration of a method. Instead, a method can have multiple throws-documentation elements (which in general are XML elements, that contain the all text after a throws tag). When documenting exceptions in methods, you must first create throws documentation for each of the exceptions that will be documented for the method, and the throws documentation must contain at least one word (which will be interpreted as the exception class name) or it will be ignored during XMI examination of the XMIDocumentFactory. So basically, before documenting, create all classes, fields and methods (including arguments, return value and throws documentation) as the XMI Documenter is unable to create model elements, it can only modify existing ones.

The first thing that may seem odd is that interface and class elements found in the analyzed XMI file are both mapped to ClassElement instances and thus cannot be distinguished at runtime. The reason for this is that the parsing and handling of interfaces in the XMI file was not contained in the original design (neither designer nor design review board noticed that). When this was identified by a developer during dev phase, the PM decided that it was too late (i.e. the deadline was too near) to make any changes to the public API of the XMI documenter component – so the least intrusive fix was introduced – mapping interfaces to ClassElements also. Perhaps a 1.1 component will fix this issue.

One more thing worth mentioning is that the XMI Documenter component does not have the scope of validating given XMI files. It may continue and work on an XMI file that is syntactically invalid but is valid enough in structure and content to be parsed ant modified by the component – so this basically means Garbage in-Garbage Out, i.e. the component will not always detect invalid XMI documents.

The third detail that is worth noticing is that when the XMIDocument instance is constructed using the method XMIDocumentFactory#createDocument(org.w3c.dom.Document), the backing DOM used by the created XMIDocument instance is the one given as argument to that create method – so modifications to the XMI document result in modifications of the DOM document and vice versa. In the worst case that means that external modifications of the DOM document may break the integrity of the XMIDocument instance and the result of the file written out by XMIDocument#writeTo may be unpredictable. So keep that in mind when using that particular factory method.

4. How to setup and run the XMI Documenter UI demo

The demo provided along with this article contains the class mentioned in the article and a build script to compile and run the editor. This demo can either be used as-is or as a starting point for your own modifications and improvements to the XMI editor. Most of this chapter describes the setup of dependencies of XMI Documenter.

4.1 TopCoder Software Components used

- GUID Generator 1.0
- Base Exception 1.0
- XMI Documenter 1.0

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.

4.2 Third Party Components used by XMI Documenter

- JAXP (Required only with Java 1.4) : version 1.3.1 : https://jaxp.dev.java.net/servlets/ProjectDocumentList?folderID=4584&expandFolder=4584&folderID=0

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location.

4.3 Dependencies Configuration

Follow the instructions at https://jaxp.dev.java.net/Updating.html, section "Using JAXP with version 1.4 of the Java 2 platform" to configure JAXP 1.3 for use with Java 1.4. JAXP 1.3 is already included in Java 1.5, so nothing needs to be done for a Java 1.5 environment.

In General the following steps are required:

- Download Xalan-J 2.7.0 (http://xml.apache.org/xalan-j/) from http://www.apache.org/dyn/closer.cgi/xml/xalan-j
- All five Jars from the binary 2-Jar distribution are needed.

Now two options exist depending on whether JAXP-1.3 shall be integrated in the JDK or only be used when needed:

- Integration in JDK 1.4:
    - Put the Jars in `%JDK_HOME%/jre/lib/endorsed` – Create that directory if it doesn't exist
    - Jars in that directory are loaded by the JDK classloader before rt.jar is loaded, i.e. the files are prepended in the JVM boot class path
- Use the Jars only when needed
    - For compilation use the `javac`-argument `–Xbootclasspath/p:xml-apis.jar` (where `xml-apis.jar` is the path to the `xml-apis.jar` from the Xalan binary distribution)
    - At runtime use the JVM argument `–Xbootclasspath/p:` and mention all 5 jars from the xalan distribution

- Actually the build file is aware of whether the JAXP 1.3 must be prepended to the boot class path and does so during compile and run, so under normal conditions only `ant startdemo` must be executed and the Jars are automatically added to the boot class path of compiler and JVM if JVM version is not 1.5

Running the demo

To run the demo GUI simply call `ant startdemo`. The window shows three buttons, load can be used to load an XMI file (*.zuml files must be unzipped before editing), then while navigating through the tree you can view and edit the documentation of the selected node in the edit area on the right hand side. At any time the current state of the XMI document can be saved using the save button. The third button highlights tree nodes in red if they either contain no documentation or if they contain children with no documentation.