JQTI-Lite Library:

DEVELOPER'S GUIDE

GTI – Interactive Technologies Group
Universitat Pompeu Fabra, Departament de Tecnologia
Pg. De Circumval·lació 8, E-08003 Barcelona (Spain)
http://www.tecn.upf.es/gti

Table of contents

1 BASIC STRUCTURE OF THE LIBRARY.......... 2

2 REFERENCES 5

3 SAMPLE CODE SNIPPETS 6

3.1 SNIPPET 1 6
3.2 SNIPPET 2 6
3.3 SNIPPET 3 7

Introduction

jQTI-Lite is a Java library for managing the IMS Question & Test Interoperability Lite specification (QTI Lite). The library supports a Questions and Assessments authoring tool, QAed (Questions and Assessments Editor). Both have been developed in the context of a European Union supported project called SCOPE (Structuring Content for Online Publishing Environments)[1], and follow the Free Software pattern.

We intend to continue our developments related to eLearning systems. Both tools and additional material are available at our open web initiative Learning Technology Open Source[2]. Our main focus is the development of a set of libraries and applications compliant with the IMS specifications[3].

Contact

We look forward to your feedback. Please contact Josep Blat, GTI director at **josep.blat@upf.edu** or Juanjo Martínez, software engineer and developer at **juanjo.martinez@upf.edu**

[1] www.tecn.upf.es/scope
[2] www.tecn.upf.es/gti/leteos
[3] www.imsproject.org

1 Basic structure of the library

It is important to know the philosophy and the basic architecture of this library to work well with it. The first step to understand jQTI-Lite, and to be able to use it, is to be familiar with the IMS Question & Test Interoperability specification. Otherwise, our library is completely unintelligible. The home of IMS Question & Test Interoperability Lite is at **http://www.imsproject.org/question/index.cfm**. There you can find the essential information about this specification (documents, XML Schemas, etc.).

Our library is Java based. It is based on the JDOM and Xerces libraries, which act as the XML input and output engine. Our library offers a set of classes for managing data structures that map the IMS QTI-Lite specification and deals for its XML serialization and parsing.

The main goal of the library is to provide the basic functionality to support the development of applications based on the QTI-Lite IMS specification. The library acts as the kernel of the applications, such as our QAed, previously mentioned.

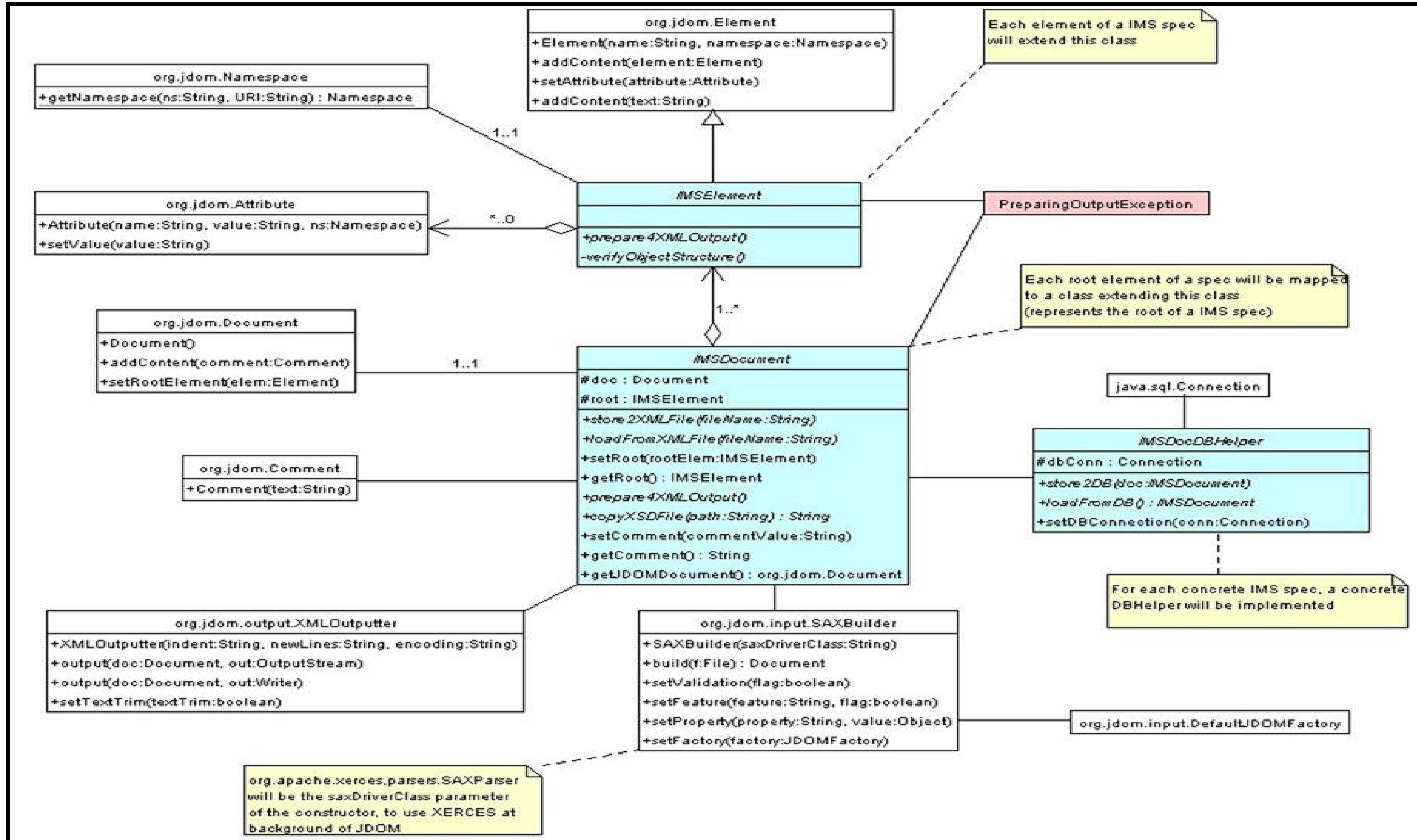
The JavaDoc documentation delivered both in the source code and the built version that you can find in the *Learning Technology Open Source* website provides specific details about the classes (methods, member variables, static constants, etc.) of the library.

At the following page a UML diagram illustrates the basic architecture of the library. In the diagram, blue boxes represent abstracts classes, red boxes represent exception classes, white boxes represent normal classes and yellow boxes are annotations.

The architecture is based on two Java libraries for XML: JDOM, which is the main element of the structure, and Apache Xerces, which supports JDOM (for example performing the validation of a XML file against a XML Schema in a parsing operation).

Each IMS specification is a hierarchy of XML elements: our approach is to map directly this XML hierarchy to a hierarchy of Java classes. Each element of the QTI-Lite specification is mapped into a Java class on our library; the attributes and child elements of each specification element are mapped to member variables with its "getter" and "setter" methods in the Java class.

The *IMSElement* class acts as an abstraction for each class representing an element of the specification. Thus every class of this type must extend the IMSElement abstract class. The IMSElement class extends the JDOM element, making all the classes which map specification elements JDOM elements too.


org.jdom.Element
+Element(name:String, namespace:Namespace)
+addContent(element:Element)
+setAttribute(attribute:Attribute)
+addContent(text:String)
Each element of a IMS spec will extend this class
org.jdom.Namespace
+getNamespace(ns:String, URI:String) : Namespace
1..1
org.jdom.Attribute
+Attribute(name:String, value:String, ns:Namespace)
+setValue(value:String)
*..0
IMSElement
+prepare4XMLOutput()
-verifyObjectStructure()
PreparingOutputException
Each root element of a spec will be mapped to a class extending this class (represents the root of a IMS spec)
1..*
org.jdom.Document
+Document()
+addContent(comment:Comment)
+setRootElement(elem:Element)
1..1
IMSDocument
#doc : Document
#root : IMSElement
+store2XMLFile(fileName:String)
+loadFromXMLFile(fileName:String)
+setRoot(rootElem:IMSElement)
+getRoot() : IMSElement
+prepare4XMLOutput()
+copyXSDFile(path:String) : String
+setComment(commentValue:String)
+getComment() : String
+getJDOMDocument() : org.jdom.Document
java.sql.Connection
IMSDocDBHelper
#dbConn : Connection
+store2DB(doc:IMSDocument)
+loadFromDB() : IMSDocument
+setDBConnection(conn:Connection)
For each concrete IMS spec, a concrete DBHelper will be implemented
org.jdom.Comment
+Comment(text:String)
org.jdom.output.XMLOutputter
+XMLOutputter(indent:String, newLines:String, encoding:String)
+output(doc:Document, out:OutputStream)
+output(doc:Document, out:Writer)
+setTextTrim(textTrim:boolean)
org.jdom.input.SAXBuilder
+SAXBuilder(saxDriverClass:String)
+build(f:File) : Document
+setValidation(flag:boolean)
+setFeature(feature:String, flag:boolean)
+setProperty(property:String, value:Object)
+setFactory(factory:JDOMFactory)
org.jdom.input.DefaultJDOMFactory
org.apache.xerces.parsers.SAXParser will be the saxDriverClass parameter of the constructor, to use XERCES at background of JDOM

The *IMSDocument* class acts as an abstraction for a concrete document of the specification, in other words it encapsulates the hierarchy of a concrete document which follows the specification. This encapsulation is done attaching a root element to the document (obviously the attached element must contain all the hierarchy of the specification which is following).

The structure for performing the XML serialization (output of XML files) is based on the *Chain of Responsibility* object-oriented pattern. The philosophy of this pattern is that each object in a hierarchical object-oriented model is responsible for its own behaviour and operations, and is responsible for the execution of the same operation in the following object of the "chain". This pattern is present in the abstract methods *prepare4XMLOutput* and *verifyObjectStructure* in the *IMSDocument* and *IMSElement* classes and they are the basis for the XML serialization.

The *prepare4XMLOutput* method of the *IMSDocument* class executes the same method of the root element attached. The root element will load its JDOM element part because the serialization is done by JDOM and it only works with *Element* classes of the JDOM library.

The *verifyObjectStructure* method checks the integrity of a concrete object reporting a *PreparingOutputException* in case the object is not well formed. After the execution of the *prepare4XMLOutput* of the root, the latter will execute the same method of its sons in the "chain"; then the sons will execute their sons' method until the "chain" arrives to an ending class.

The structure for performing XML parsing is free: it is not forced by any abstract class. The parsers we have implemented use JDOM and Xerces for the validation and parsing tasks. After these tasks are performed, the information in the XML DOM tree generated in the parsing process done by JDOM is transformed by the parser into specific library objects.

2 References

McLaughlin, Brett.: *Java & XML, 2nd Edition. Solution to Real-World Problems.* O'REILLY (2001). ISBN: 0-596-00197-5.

Rusty Harold, Elliotte.: *Processing XML with Java: A guide to SAX, DOM, JDOM, JAXP, and TraX*. Pearson Education (2003). ISBN: 0-201-77186-1.

Gamma, Erich et al.: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley (1994). ISBN: 0-201-63361-2.

3 Sample code snippets

3.1 Snippet 1

```
import java.io.IOException;
import java.io.File;

import es.upf.ims.qtilite.parser.QTILiteParser;
import es.upf.ims.qtilite.QTILiteDocument;
import es.upf.ims.PreparingOutputException;
import es.upf.ims.ParsingException;
import es.upf.ims.qtilite._LIBGlobal;

public class SnippetOne {

    static final String SOURCE_FILE = "C:\\Tests\\SampleQTILite.xml";
    static final String DESTINATION_FILE = "C:\\qti_result.xml";

    public static void main(String[] args) {
        try {
            QTILiteDocument qtidoc = new QTILiteDocument();
            qtidoc.loadFromXMLFile(new File(SOURCE_FILE));
            qtidoc.setComment("Generated by SnippetOne");
            qtidoc.prepare4XMLOutput();
            qtidoc.store2XMLFile(new File(DESTINATION_FILE));
        }
        catch(ParsingException pe) { pe.printMessage(); }
        catch(PreparingOutputException poe) { poe.printMessage(); }
        catch(IOException ioe) { System.out.print(ioe.getMessage()); }
    }
}
```

This code loads a QTI-Lite XML file first. After the *QTILiteDocument* object is loaded then the same object is stored into another QTI-Lite XML file.
Remark that *prepare4XMLOutput* is invoked before *store2XMLFile* is. As it was mentioned previously, the goal of this method is to build the XML DOM tree of the QTI encapsulated in the set of objects that form the *QTILiteDocument*. This method is mandatory and necessary before storing a *QTILiteDocument* into a QTI-Lite XML file.

3.2 Snippet 2

```
import java.io.IOException;
import java.io.File;

import es.upf.ims.qtilite.QuesTestInterop;
import es.upf.ims.qtilite.Item;
import es.upf.ims.qtilite.QTILiteDocument;
import es.upf.ims.PreparingOutputException;
import es.upf.ims.qtilite._LIBGlobal;

public class SnippetTwo {

    static final String DESTINATION_FILE = "C:\\qti_result.xml";

    public static void main(String[] args) {
        try {
```

```
            QuesTestInterop root = new QuesTestInterop();
            Item item = new Item("An_Item");
            root.addItemElement(item);

            QTILiteDocument qtidoc = new QTILiteDocument(root);
            qtidoc.setComment("Generated by SnippetTwo");
            qtidoc.prepare4XMLOutput();
            qtidoc.store2XMLFile(new File(DESTINATION_FILE));
        }
        catch(PreparingOutputException poe) { poe.printMessage(); }
        catch(IOException ioe) { System.out.print(ioe.getMessage()); }
    }
}
```

This code shows the construction of a simple *QTILiteDocument* object. The *QuesTestInterop* is the class that encapsulates the root element of the QTI-Lite specification, which is *questestinterop*. Remark that the *QTILiteDocument* object is created by attaching the *QuesTestInterop* root element on the constructor to it.

This code shows how our library allows to build the hierarchy of a QTI-Lite data model in a simple way.

3.3 Snippet 3

```
import java.io.File;
import java.util.List;

import es.upf.ims.qtilite.parser.QTILiteParser;
import es.upf.ims.qtilite.QuesTestInterop;
import es.upf.ims.qtilite.Item;
import es.upf.ims.qtilite.QTILiteDocument;
import es.upf.ims.ParsingException;
import es.upf.ims.qtilite._LIBGlobal;

public class SnippetThree {

    static final String SOURCE_FILE = "C:\\Tests\\SampleQTILite.xml";

    public static void main(String[] args) {
        try {
            QTILiteDocument qtidoc = new QTILiteDocument();
            qtidoc.loadFromXMLFile(new File(SOURCE_FILE));
            QuesTestInterop root = (QuesTestInterop)qtidoc.getRoot();
            List list = root.getItemElements();
            if(list != null) {
                String itemIdent = ((Item)list.get(0)).getIdent();
                System.out.println("Item identifier: " + itemIdent);
            }
        }
        catch(ParsingException pe) { pe.printMessage(); }
    }
}
```

This code shows how to get data from a QTI-Lite data model based on our library. This code loads a *QTILiteDocument* from a QTI-Lite XML file. After loading, the code goes through the hierarchy of objects until an *Item* and gets a data value.