DANGEROUS GOODS MANIFEST SUBMISSION

WITH EBXML MESSAGE SERVICE

(XMLDG)

for

The MARINE DEPARTMENT

VOCABULARY LIBRARY USER GUIDE

Version: 1.0

October 2003

COPYRIGHT

LIMITATION OF LIABILITY

The contents of this User Guide are provided for the information and use of all users of the Electronic Submission of Dangerous Goods Manifests Service ("the Service"), including vessel owners, agents and masters. All information contained in this User Guide is believed to be accurate but neither the copyright holder nor any person or organisation concerned in the preparation or publication of this User Guide accepts any liability of any nature whatsoever for any loss suffered either directly or indirectly as a consequence of the use by users of the Service or this User Guide or any trading activity flowing therefrom.

For full details of legal requirements related to the Service, it should refer to the section 4 of the Dangerous Goods (Shipping) Regulations.

TABLE OF CONTENTS

1. INTRODUCTION

The XMLDG vocabularies library provides an XML binding framework for the XMLDG project. It maps each element in the XML schema used in the project to a Java class. Each Java class provides getter and setter methods to manipulate and parse the XML documents.

The main advantage of XML binding framework is that the developers of the applications need not touch any XML programming to develop the applications. Nowadays all common programming models for XML such as DOM, SAX or JDOM, are error-prone because the application needs to pass the names of the elements to the API of the XML libraries. The misspelling of the elements names will cause bugs in the application but they cannot be checked at compile-time.

By binding the XML schema to Java classes, there are no misspelling problems because the methods are restricted by the methods signatures of the vocabulary library.

There are two main specifications that restrict the syntax and semantics of the XML messages in the project: XML Schema Specification (SS) and Implementation Instruction (II). The restrictions of SS are implemented in the form of XML Schema. On the other hand, the constraints imposed by II are configurable. The configurations are in the form of a XML document. In the following paragraphs, the restrictions imposed by II are referred to constraints configurations.

2. INSTALLATION

In the following part, `<XMLDG_HOME>` refers to the directory of extracted XMLDG vocabularies `library` distribution.

1. Add `<XMLDG_HOME>/build/xmldg.jar` and all the libraries in lib directory to the `CLASSPATH`.
2. Copy `<XMLDG_HOME>/conf/vocab.properties` to your home directory.

   If you run the application on MS Windows NT/2000/XP, the home directory will be `"C:\Documents and Settings\<your_username>"`. For Windows 9x, the home directory is the directory of Windows.

   If you deploy the application on Tomcat running as Windows NT services, you need to `take` special care about the home directory. By default, the "Apache Tomcat" service runs with the logon user `"LocalSystem"`. Its home directory is `"C:\Documents and Settings\Default User"`. You should either change the user for running the Tomcat service, or put the `vocab.properties` in `"C:\Documents and Settings\Default User"`.

3. Edit `vocab.properties` in the home directory. You are recommended to change the value of `hk.hku.cecid.phoenix.vocab.dg.dir and hk.hku.cecid.phoenix.vocab.constraints` only:

Property	Meaning
`hk.hku.cecid.phoenix.vocab.dg.dir`	The directory containing the XML Schemas for dangerous goods manifest and constraints
`hk.hku.cecid.phoenix.vocab. appendOptionalChildren`	Whether optional children are appended when a new element is created. It takes true or false. If you set it to `true`, and when you create a new instance of a document, you need not call the `createXXXX()` methods to create instances of optional children elements. The factory value is set to `false`.
`hk.hku.cecid.phoenix.vocab.constraints`	The path of the constraints configurations. Bases on the constraints configurations, the library checks whether the documents satisfy the restrictions in II.

4. Check the installation by running `<XMLDG_HOME>/sample/run.bat`. If the installation is successful, you should be able to see this output:

```
/DangerousGoodsManifest/DocumentHeader/DocumentTypeCode:
DGM
/DangerousGoodsManifest/DangerousGoodsItem/GrossMassMeasu
re: KGM
/DangerousGoodsManifest/DocumentHeader/DocumentId:
abcdefghijabcdefghijabcdefghi
jabcde
=============== DangerousGoodsItem 1
======================
/DangerousGoodsManifest/DangerousGoodsItem/DangerousGoods
/FlashpointMeasure: 111
.1
=============== DangerousGoodsItem 2
======================
/DangerousGoodsManifest/DangerousGoodsItem/DangerousGoods
/FlashpointMeasure: 112
.1
=============== DangerousGoodsItem 3
======================
/DangerousGoodsManifest/DangerousGoodsItem/DangerousGoods
/FlashpointMeasure: 113
.1
```

The batch actually runs the sample program `DangerousGoodsManifest.java`, which creates a `DangerousGoodsManifest` object, writes it to disk (`DangerousGoodsManifest.xml`), and parse it.

3. BASIC USAGE

3.1 CREATING NEW XML DOCUMENTS

To create a new document, you can call the method `newInstance` of the class representing the document. For example, the following code creates a new instance of `DangerousGoodsManifest`:

```
DangerousGoodsManifest dgm =
DangerousGoodsManifest.newInstance();
```

When creating a new instance of a document, all the mandatory children restricted by XML Schema (but not constraints configurations), such as `DocumentId` of `DangerousGoodsManifest`, will be automatically appended. Therefore, you may access `DocumentId` in this way:

```
DocumentId documentId = dgm.getDocumentId();
```

In the above code fragment, the `documentId` variable will not be null even if you never explicitly append `DocumentId` to `DangerousGoodsManifest`. It is because `DocumentId` is a mandatory child of `DangerousGoodsManifest` in the XML Schema.

However, whether the optional children are appended when creating a new instance of document depends on the value of the property `hk.hku.cecid.phoenix.vocab.appendOptionalChildren` in the `vocab.properties`. If the value is `false`, the optional children will not be appended. Your application needs to explicitly create the children and append them to an element. If the value is `true`, exactly one child for each child element will be automatically appended. The factory value of this property is `false`.

3.2 CREATING NEW CHILD ELEMENT

Each non-leaf element has a `createXXXX` method, where `XXXX` is the name of its child element, to create a new instance of a child element. For example, the following code create a new instance of `DangerousGoodsItem`:

```
// The variable dgm is DangerousGoodsManifest
DangerousGoodsItem item = dgm.createDangerousGoodsItem();
```

After creating the child element, you need to call the setter or adder methods to append the child element to an element. For details, please see the section 3.3 and 3.4.

In the API documentation, you may see that both `createXXXX` method of parent element and `newInstance` method of child element can return a new instance of an element. For example, there are two ways to create a `DocumentId`:

```
// The variable dgm is DangerousGoodsManifest
DocumentHeader documentHeader = dgm.createDocumentHeader();
…
…
DocumentId documentId = documentHeader. createDocumentId;
DocumentId documentId2 = DocumentId.newInstance();
```

The difference is that the `createXXXX` method creates an element that is under the context of its parent element. This context affects the constraints checking of the child element. If we create the element with `newInstance` method, the created element is supposed to be the root element of the document.

In the above example, if there is a constraint configuration restricting that the element of path `/DangerousGoodsManifest/DocumentHeader/DocumentId` must have a text of length 1 to 35, this constraint is only applied on `documentId` but not `documentId2`, which is supposed to be a root element.

Normally whenever you would like to create a root element such as `DangerousGoodsManifest`, `Acknowledgement` and `Credential`, you should use the `newInstance` method. When creating children elements, you should use `createXXXX` method. The reason of creating `newInstance` method for each element is that it provides flexibility and compatibility with the XML Schema, which allows all elements are root elements.

3.3 SETTING OR REPLACING CHILD ELEMENT

Each non-leaf element has a `setXXXX` method, where `XXXX` is the name of its child element, to set or replace the child element. For example, the following code sets or replaces the `Transport` of a `DangerousGoodsManifest`:

```
// The variable dgm is DangerousGoodsManifest
Transport transport = dgm.createTransport();
// Setting the content of the transport
…
…
…
dgm.setTransport(transport);
```

If the cardinality of the child element is more than one, the `setXXXX` method will have an index parameter. The index parameter is to indicate which child element should be replaced. If such element at the specified index does not exist, `VocabException` will be thrown.

3.4 APPENDING CHILDREN ELEMENTS

If an element has a child whose cardinality is more than one, there is an `addXXXX` method for the child element, where `XXXX` is the child element name. For example, the following code is to add a `DangerousGoodsItem` to a `DangerousGoodsManifest`:

```
DangerousGoodsItem item = dgm.createDangerousGoodsItem();
// Setting the content of the DangerousGoodsItem
…
…
…
// The variable dgm is DangerousGoodsManifest
dgm.addDangerousGoodsItem(item);
```

The number of children elements for an element is restricted by SS and II.

3.5 SETTING LEAF-NODE VALUES

Each leaf-node, such as `DocumentId`, has a method `setContent`. It allows you to set the text under the leaf-node. For example, the following code is to set the text under the `DocumentId` element:

```
documentId.setContent("abcdefghijabcdefghijabcdefghijabcde");
```

For leaf-nodes restricted with decimal type, the `String` parameter of the `setContent` method must be decimal. Otherwise, `VocabException` or `ConstraintViolatedException` will be thrown. If the type of the value is restricted in XML Schema, `VocabException` will be thrown. If there is no type restriction on the value on the XML Schema, but the type restriction comes from constraints configuration, `ConstraintsVioldatedException` will be thrown.

For example, the text under `Quantity` must be a decimal. The following code will throw `VocabException`:

```
// The variable quantity is a Quantity element
quantity.setContent("abc");
```

because the XML Schema restricts that the text value under `Quantity` element must be a decimal.

3.6 REMOVING CHILDREN ELEMENTS

Each non-leaf element may have `removeXXXX` method, where `XXXX` is the name of the child element, to remove the optional children elements. If the children elements are mandatory, there is no `removeXXXX` method for the mandatory children. For example, the following code is to remove all the `DangerousGoodsItem` of `DangerousGoodsManifest`:

```
// The variable dgm is DangerousGoodsManifest
BOOLEAN SUCCESSFUL = DGM.REMOVEDANGEROUSGOODSITEM();
```

3.7 MARSHALLING

After creating a document and populating the content by the methods mentioned above, you need to marshal the document object to a physical XML document. Each class provides a method `marshal`, which takes a `Writer` as parameter. For example, the following code is to marshal a `DangerousGoodsManifest` object the file `DangerousGoodsManifest.xml`:

```
FileWriter fw = new FileWriter("DangerousGoodsManifest.xml");
// The variable dgm is DangerousGoodsManifest
dgm.marshal(fw);
```

When marshalling, all the restrictions imposed by XML Schema and constraints configuration have to be satisfied. Otherwise, `VocabException` or `ConstraintsViolatedException` will be thrown. If the restrictions imposed by constraints configuration are violated, `ConstraintsVioldatedException` will be thrown. If the restrictions imposed by XML Schema are violated, `VocabException` will be thrown.

3.8 UNMARSHALLING

Unmarshalling is the reverse of marshalling: it is to instantiate an object representing the document by parsing a physical XML document. It is the first action to take before accessing the contents of a XML document. Each class representing an element has a `static` method `unmarshal` to do unmarshalling. The `unmarshal` method takes a `Reader` as parameter. For example, the following code is to unmarshal a `DangerousGoodsManifest`:

```
FileReader fr = new FileReader("DangerousGoodsManifest.xml");
DangerousGoodsManifest dgm =
DangerousGoodsManifest.unmarshal(fr);
```

When unmarshalling, all the restrictions imposed by XML Schema and constraints configuration must be satisfied. Otherwise, `VocabException` or `ConstraintsViolatedException` will be thrown. If the restrictions imposed by

constraints configuration are violated, `ConstraintsViolatedException` will be thrown. If the restrictions imposed by XML Schema are violated, `VocabException` will be thrown.

3.9 GETTING CHILDREN ELEMENTS

Each non-leaf element has a `getXXXX` method for each child element, where `XXXX` is the name of the child element, for you to get the children elements. The getter method returns an object representing the child element. For example, the following code is to get the `DocumentHeader` of `DangerousGoodsManifest`:

```
// The variable dgm is DangerousGoodsManifest
DocumentHeader documentHeader = dgm.getDocumentHeader();
```

If the cardinality of the children elements is more than one, a `List` of the objects representing the children elements is returned. However, if such children do not exist, an empty `List` is returned instead of null. For example, the following code is to get all the `DangerousGoodsItem` of `DangerousGoodsManifest`:

```
// The variable dgm is DangerousGoodsManifest
List items  = dgm.getDangerousGoodsItem();
Iterator itemsIter = items.iterator();
While(itemsIter.hasNext()) {
   DangerousGoodsItem item =
(DangerousGoodsItem)itemsIter.next();
   // Do whatever you like…
  …
 …
}
```

3.10 GETTING LEAF-NODE VALUES

Similar to setting leaf-node values, the leaf-node elements provide `getContent` method to get the text under the elements. For example, the following code is to get the text value under a `Quantity` element:

```
// The variable q is Quantity
String q = quantity.getContent();
```

4. DEFAULT ELEMENT VALUES AND OCCURRENCES

When an element is created, the textual content value is set according to the following rules:

- If there is a Content Constraint in XMLDG II for the element, the fixed value is set.
- For elements with Attribute Constraint in II, the fixed attribute and its value are set.
- For elements with fixed enumeration values both in SS and II, the first value of the enumeration is set. The II overrides the SS.
- If the above rules do not apply, empty string is set for element with String type; "0" is set for element with Integer type; "0.0" is set for element with Decimal type; current date-time is set for element with dateTime type. The data types are the data types in the SS.

When an element is created, the number of children elements is determined by the following rule:

- If it is a mandatory child specified in the SS, the library creates the child with its occurrence according to the SS.
- If it is an optional child, the library looks for the `hk.hku.cecid.phoenix.vocab.appendOptionalChildren` attribute in the properties file `vocab.properties`. If the property is set to true, one element is created for each optional child and appended to the parent. Otherwise, optional children will not be appended. It is suggested to set the property to be false.
- The Occurrence Constraint in II will not affect the default number of children elements when creating an element.

5. EXCEPTIONS

The vocabulary library has five exceptions, in addition to those defined in JDK, which may be thrown by the vocabulary library. They are:

- `VocabException`
- `SchemaNotFoundException`
- `ConstraintsNotFoundException`
- `ConstraintViolatedException`
- `InvalidConstraintException`

The `VocabException` is the base class of the other four exceptions. Therefore, if your application has the same error handling no matter which exception is thrown, you can just catch the `VocabException`:

```
try {
      // Your code for composing or parsing
      // DangerousGoodsManifest or other documents
      //
}
catch (VocabException e) {
      // Do exception handling here
}
```

5.1 VOCABEXCEPTION

It is a root class of the exceptions thrown by the library. It will be thrown if:

1. The document does not satisfy SS during unmarshalling. If the document was composed by this library, it will not happen normally because the library will not allow you to marshal a document that does not satisfy SS.
2. More than maximum allowable number (specified by the SS) of children elements is added to an element.
3. The textual content of a leaf node is set by `setContent` method but the content violates the schema/specification
4. An element is added twice to another element
5. The constraints do not satisfy the schema for the constraints language.
6. The dependent libraries throw an unexpected exception or error.

5.2 SCHEMANOTFOUNDEXCEPTION

It indicates that the XML Schema for SS (excluding the XML Schema for the constraints configuration) cannot be found. This exception may be thrown when calling the `newInstance` method or the `unmarshal` method, which needs the

XML Schema for validation of XML documents such as `DangerousGoodsManifest`.

To solve this problem, you should make sure that the value of the properties `hk.hku.cecid.phoenix.vocab.dg.dir` in `vocab.properties` is the directory containing all the XML Schemas and the XML Schema files exists.

5.3 CONSTRAINTSNOTFOUNDEXCEPTION

It indicates that the constraints configurations cannot be found when unmarshalling or creating new instance of an element or a document (by calling `newInstance` method).

To solve this problem, you should make sure that the value of the property `hk.hku.cecid.phoenix.vocab.constraints` is the path of the constraints configurations.

5.4 CONSTRAINTVIOLATEDEXCEPTION

It indicates that one or more constraints in II is/are violated. This exception may be thrown when marshalling, unmarshalling, creating new instances of documents, adding children elements, or setting the textual content of an element.

This exception has these methods to help you to solve the problem if it is thrown:

- *getMessage* -
  It tells you what type of constraint has been violated by your application. The path of the element violating the constraint is also contained in the message.
- *getDocumentId* –
  It tells you that the `DocumentId`, which is contained in `DocumentHeader`, of the document violating any constraints in constraints configurations. Note that this method will only return non-null value if the violating element is added to a document containing `DocumentId` (i.e. `DangerousGoodsManifest` and `Acknowledgement`).
- *getPath* –
  It returns the path of the violating element.
- *getTypeCode* –
  It returns the code of the constraint violated. The possible type codes can be found in the Implementation Instruction.

5.5 INVALIDCONSTRAINTEXCEPTION

It indicates that invalid constraints exist in the constraints configurations. This exception may be thrown when unmarshalling and creating new instance of documents (by calling `newInstance` method).

The `getMessage` method of this exception tells you the details. You can simply follow the message to correct the improper configuration. For details of configuring the constraints, please see section 4.

5.6 EXCEPTIONS CATCHING

`VocabException` is the base class of all other four exceptions and each of the other four exceptions directly extend `VocabException`. From programming perspective, your application can simply catch `VocabException` to satisfy the compiler requirement. Such catching block is responsible for handling all types of exceptions. However, in many cases, you may want to have different exception handling for each type of exceptions. In all cases, `VocabException` should be the last exception in the catch block as it is the base class for the other exceptions.

Generally speaking, to consider how to catch the exceptions, you should consider firstly that the application should only handle user-level problems, or both user-level problems and configuration problems.

Let us firstly consider how we should catch the exceptions for the case that the application is supposed to handle user-level problems. In many cases, we should only alert the user for constraints violation. We do special exception handling when constraint violation happens. All the other exceptions, such as `SchemaNotFoundException` or `InvalidConstraintsException`, are considered as configurations problems. If you want the users to be aware of constraints violation but not the details of configuration problems, your application should catch the exceptions with this exceptions catching block:

```
try {
  // populating the content of a document, marshalling and
  // unmarshalling
}
catch(ConstraintsViolatedException e) {
  // Notify the users, notify upper level of applications by
  // re-throwing, and also do some clean-up if needed
}
catch(VocabException e) {
  // Notify the users that some configuration problems here.
  // Do some clean-up if needed
}
```

On the other hand, if you want the users to be aware of the details of configurations problems besides user-level problems, or do special handling for a particular type of configuration problem, you should catch other types of configuration-related exceptions explicitly. Exhaustively talking about all combinations maybe a little bit

tedious and not very meaningful. Let us consider an example. Consider that you want the user to be aware of `SchemaNotFoundException`, and you want the user to provide correct location of the XML Schema. In this case, your application should catch `SchemaNotFoundException`. Within the catching block for `SchemaNotFoundException`, you should write the code for asking the correct location of the XML Schema and change the `vocab.properties`:

```
try {
  // populating the content of a document, marshalling and
  // unmarshalling
}
catch(ConstraintsViolatedException e) {
  // Notify the users, notify upper level of applications by
  // re-throwing, and also do some clean-up if needed
}
catch(SchemaNotFoundException e) {
  // Notify the users, ask for correct location of XML Schema,
  // modify vocab.properties
}
catch(VocabException e) {
  // Notify the users that some other configuration problems
  //here. Do some clean-up if needed
}
```

6. SAMPLE CODE

A sample code, *DangerousGoodsManifest.java*, can be found in the sample directory. It creates a new instance of `DangerousGoodsManifest`, populate the contents and marshal it into a file with filename *DangerousGoodsManifest.xml*. Then, it unmarshals the file. This sample code can be used to test the installation.

7. REFERENCES

[regexp] There are two regular expressions tutorials:
http://www.gnu.org/manual/gawk/html_node/Regexp.html
http://www.zvon.org/other/PerlTutorial/Output/index.html