Xotics Core API

Developer's Guide

Product Version : 4.1.23

Version : 2.012
Date : 2006 March 2
Author : Virtual Weaver Interactive

This document is intended to provide, to the developer, a detailed description on how to use Xotics Core API, to create powerful Java applications integrating XML technologies, not just as an exchange format, but also as a new way of structuring programs.
The main purpose of Xotics Core API is to provide all necessary interfaces and classes to implement XML dialect in Java, where each XML element is represented by a JavaBean. Core API provides also additional features such as, full XPAth 2 support, document loading from various sources and saving to various destinations, thread-safe management, ...
This document is a complement to the API Javadoc. All chapters excepted the last one are dedicated to the API packaged in xotics_core410.jar file. The last chapter introduces Editing Extension API packaged in xotics_editing410.jar.

Chapter 1 : Overview

The main Xotics feature consists in mapping every XML concept into Java objects, as in the table below:

XML	Java
Dialect, identified by namespace	Xotics Data-Model (.Jar file or single DMDL document (XML dialect : Data Model Definition Language)
XML Document	XoDMInstance, java class representing an XML document
XML Element	JavaBean implementing XoObject base or derived interface, depending upon XML element kind
XML Attribute	JavaBean property, also accessible by XoProperty wrapper class

Table 1 : mapping of XML concepts into Java objects

1.1 XML dialect implementation

Xotics operates on XML dialect basis, whom it loads, when needed, the implementation in its environment. In Xotics terminology, an XML dialect implementation is called a Data-Model. A Data-Model contains all Java classes representing the dialect and an XML document in DMDL format, mainly describing :

- namespace URI identifying the dialect,
- every XML element, with its JavaBean representation,
- in option for each JavaBean, its Content-Model and some information on published properties,
- other general purpose information, such as PropertyEditor to use, additional XSD datatypes and new Xpath2 functions.

A dialect implementation can be deployed in two ways :

- A Jar file containing both code and DMDL document,
- A single DMDL file, referencing classes accessible by any classical Java manner.

Note : unlike DOM, Xotics is unable to deal with an XML dialect as long as it has not loaded its DMDL description. Moreover, all XML elements of a document must be identified by their namespace. However, Xotics provides an automatic dialect loading mecanism, and a future version will allow to create a dialect implementation «on the fly» from a DTD or XSD document (XML Schema Definition).

1.2 XO objects

In Xotics terminology, XO means «XML Object». An XO object is the JavaBean representation of an XML element. Xotics provides several kinds of XO objects, each corresponding to a specific form of element. To be XO, a Java class must be first JavaBean compliant, especially about :

- its property accessors,
- its propertyChangeEvent event support, and if possible VetoableChangeEvent too,
- its PropertyEditor and Customizer.

Next, an XO JavaBean must implement one or several XO Java interfaces, each of these defining a specific kind of XML element. Every interface is or derives from XoObject base interface, which contains common features. The table below describes every kind of XML element and its corresponding XO interface :

XML element type	Implemented XO interface
element with EMPTY Content-Model	XoObject
element with non EMPTY Content-Model	XoContainer (and XoObject by derivation)
possibly Root Element	XoRoot (and XoContainer, XoObject by derivation)
element of kind XSD Simple Type	XoTextContainer (XoObject by derivation)
Text Node	XoText (and XoObject by derivation)

Table 2 : mapping of XML elements into Java objects

Note : any Java class can become an XO object, by deriving it and implementing appropriate XO interface. Integrating existing classes is one of Xotics major benefits.

1.3 XML document representation

In Xotics terminology, an XML document is mapped into a Data-Model instance. Corresponding Java class is XoDMInstance. From XML view, though terms are different, an XoDMInstance object can be quite considered as an XML document. From Java code view, an XoDMInstance is the container and main handler of a JavaBean tree. On its content tree, it can notably :

- add, suppress or modify any object, in a thread-safe way,
- manage «undo» feature, by historizing every modification,
- perform Xpath2 requests,
- centralize all events fired when adding/removing object or modifying a Bean property,
- save content in XML local or remote document file,
- check validity, by standard XML validation techniques or in a custom manner.

1.4 Xotics Environment

All Xotics API features are accessed by a single class, XoEnvironment, whose instanciation initializes an execution space called Xotics Environment.
Xotics Environment provides two main objects : the Registry and the Factory.

The Registry, whose Java class is XoRegistry, is responsible for these following tasks :

- loading/unloading XML dialect implementations (Xotics Data-Models),
- access to relevant informations held by each loaded Data-Model including, for instance, XO JavaBean creation for a particular XML (namespace, element) couple.

The Factory, whose Java class is XoFactory, is responsible for creating XoDMInstance objects :

- by loading XML file (from InputStream or URL) or string,
- or by integrating new or existing XO object tree.

XoEnvironment class provides also two other objects, each instanciated once for an environment :

- a unique Log Message Buffer (XoLogMessageBuffer), used by core system and avalaible to users,
- the XO ClassLoader (XoClassLoader), the Class Loader used to load every resource defined in Data-Models.

Note : It's possible to instanciate in a JVM any number of XoEnvironment objects. Each instance is isolated from each other and so, can not communicate with any other instance.

Chapter 2 : XO Objects

In Xotics API context, an XML dialect implementation process consists mainly in creating or deriving JavaBeans implementing one or several XO interfaces.

2.1 XoObject interface

It's the common interface to every XO object. Any XoObject implementation is a JavaBean, with possible properties representing XML attributes.
In order for a JavaBean property to be mapped into an XML attribute, it must be accessed by its two public accessors set<property> et get<property> (or is<property> for boolean types). See §2.2 Properties section for details.
The API provides a default XoObject implementation, XoObjectSupport, usable by deriving it.

2.1.1 Accessors

The following methods are accessors to fundamental XO object properties. These properties are used and set by Xotics environment. Every class implementing XoObject must declare corresponding properties to accessors described in this section, in order to store or provide these values when Xotics environment asks for.

```
public void setXmlNameSpace(String nsuri);
public String getXmlNameSpace();

public void setXmlLocalName(String localName);
public String getXmlLocalName();
```

These four methods above associate XML namespace and local name to an XO object. These two properties are writable because a single XO object class can represent distinct XML elements, in same or different namespaces. These properties are set at XO object creation time, when using the Registry. Otherwise, the developer is responsible for affecting appropriate values.

Note : when an XO object is designed to be exclusively associated to one single XML element, setters can be left empty and getters can return constant values.

```
public void setXoDMInstance(XoDMInstance dmi);
public XoDMInstance getXoDMInstance();
```

Above methods provide acces to document instance owner of XO object. When an XO object doesn't belong to any XoDMInstance object, this property is set to null.

```
public void setXoParent(XoContainer parent);
public XoContainer getXoParent();
```

An XML document is a tree. To make a document, XO objects are structured in a double-chained tree, i.e. each object knows its father. XO object's father is obviously of type XoContainer.

```
public Locale getLocale();
public void setLocale(Locale locale) throws PropertyVetoException;

public byte getXmlWhiteSpace();
public void setXmlWhiteSpace(byte wsd) throws PropertyVetoException;
```

These accessors stand for special XML attributes xml:lang et xml:space.
xml:lang value is represented by an object of type java.util.Locale. xml:space is a byte having either XoConstants.WS_PRESERVE or XoConstants.WS_COLLAPSE constant value, to reflect respectively «preserve» or «default» XML constants.
These properties are managed by Xotics environment only if they are published (via DMDL document) for root element(s) implementation(s) of a Data-Model.

2.1.2 Listeners

```
public void addPropertyChangeListener(PropertyChangeListener listener);
public void removePropertyChangeListener(PropertyChangeListener listener);
public void addVetoableChangeListener(VetoableChangeListener l);
public void removeVetoableChangeListener(VetoableChangeListener l);
```

Since an XO object is a JavaBean, it implements standard listener registering methods for PropertyChange and VetoableChange events. PropertyChange events are systematically taken in consideration by internal core system. VetoableChange events are listened by Xotics environment to protect a document in Read-Only mode. Developers must keep track of PropertyChange listeners, and should do the same for VetoableChange listeners, when Read-Only mode can be useful or simply possible.

2.1.3 Miscellaneous methods

```
public Object clone();
```

An XO object must be Cloneable.

```
public void setIntegrated(boolean integrated);
```

This method is called by internal system both when an XO objet is fully integrated to a document, and when it is fully removed from a document. Indeed, add and removal are multi-steps processes. This call informs designated object of its new state, in order for it to perfom some useful operation.

```
public boolean equalsXoObject(XoObject o);
```

This method checks whether an external XO object is equal to current object. This equality must be considered from an XML point of view. Thus, two different classes can be equal if they have same XML namespace, local name and attribute values. Implementation of this method is optional and free, but must return false by default. In case objects are also containers, equality checking must not be recursive for potential descendants, it concerns only the objects themselves.

```
public void checkXoValidity() throws XoValidityException;
```

This method contains all code suitable to check validity of an XO object. The method must exclusively check current object validity and never potential descendants (in case current object is also a container). Thus, classical checking is about property values, individually or against other values or element features somewhere in current document.

```
public boolean isXoPropertyToWrite(String pname);
```

An XO object is responsible to tell, on demand from Xotics environment, whether or not a property, specified by name, has to be written in case of saving in XML file.

2.2 Properties

From Xotics context, a JavaBean property is the Java representation of an XML attribute. To be ellectible as an XML attribute mapping, following rules must be respected :

- there must be public getter and setter methods,
- setter method must send PropertyChange event, as specified in JavaBean standard,
- the same setter method should send VetoableChange event also, in order for Xotics environment to guarantee read-only mode on a document,
- value of the property can be converted into string and created from the same string, the XML representation of this value.

This last condition needs details. In Xotics API, there is two ways to perform conversion between string and java formats. Either the Java type of the property is self-convertible, or a JavaBean PropertyEditor class is associated to the property or its java type.

2.2.1 Self-convertible property

To be self-convertible, the property Java type has specific methods : a constructor with a String as param, to create a new instance from a String value, and the method toString() which convert it into a String representation. These operations are bijectives that is, if : `Type1 xa = new Type1("xa")` then : `new Type1(xa.toString()).equals(xa)` must return true.
For instance, java.lang.Integer is self-convertible, because :

```
Integer int1 = new Integer("10");
new Integer(int1.toString()).equals(int1) // is true
```

2.2.2 PropertyEditor

For Java types that can not respect these rules above, a PropertyEditor must be associated to either the property or its Java type. This association is specified in a DMDL document (see chapter 3). In such document, a PropertyEditor class can be associated to a particular Java type : each time this type is encountered and need to be created from XML text or converted into it, a specific PropertyEditor object is used. A PropertyEditor class can be also associated to a particular property : then this PropertyEditor is used to convert this property only.
For more informations about usage of PropertyEditor in property conversion, please refer to the chapter 6.

Note : some special properties which are not self-convertible do not need PropertyEditor, such as byte value representing xml:space attribute, or java.util.Locale representing xml:lang attribute.

Even self-converted properties are internally associated to a PropertyEditor, a default one.

2.2.3 XoProperty wrapper

As representing an XML attribute, a JavaBean property can be considered as a node, especially when performing an XPath request. That's why a property wrapper has been defined, as XoProperty class, to represent each existing property instance, that is a specific property of a particular XO object instance.
Each XoProperty object is created by Xotics environment and accessible by the Registry (see chapter 3).
The main functions of XoProperty are :

- to represent each property instance as an object (an XoNode for XPath request),
- to give all needed informations about property instance, such as its java and XML names, XoObject owner, java type, etc.
- to set and get property value in a thread-safe way.

An XoProperty class implements XoNode interface, common to every object mapping an XML node. Its methods can be classified in two parts : informations and setting/getting value.

2.2.3.1 Information methods

```
public XoObject getOwnerObject();
public String getJavaName();
public String getXmlName();
public Class getValueClass();
```

Here above are the main informations relative to the wrapped property instance. A property instance is defined by the couple property name and XO object instance owner of this property. These informations are held by the wrapper and can not change for an XoProperty instance.

2.2.3.2 setting/getting methods

```
public Object getValue() throws XoException;
public String getValueAsText() throws XoException;
public void setValue(Object val) throws XoException, PropertyVetoException;
public void setValueAsText(String text) throws XoException,
PropertyVetoException;
public PropertyEditor createPropertyEditor();
```

Value can be set/get as Java object or as XML text equivalent. When corresponding property is in a DM instance, these methods are executed in a thread-safe way.
XoProperty offers the way to create an instance of PropertyEditor associated to the property. This createPropertyEditor() method never returns null, as there is a PropertyEditor for every Property, even self-converted ones (in this case, a default PropertyEditor is internally affected).

2.3 XoContainer interface

This interface adds to XoObject specific methods which make XO object a container for other XO objects. Xotics API provides a default implementation, XoContainerSupport, usable by derivation.

2.3.1 Children add and removal methods

```
public int addXoChild(XoObject child, int index) throws XoException;
```

Adds an XO object child to current XoContainer. Index argument allows insertion. If index is less than 0 or superior to current children count, child is appended to children list. The method must return effective insertion index, which can be different from index argument.
When new child is successfully added, the method must also set child's parent by calling setXoParent() method on the child.
addXoChild() must throw an XoException in case of any problem occurring during adding process, in particular when child is not welcome. This checking can be performed by calling the following method inside addXoChild().

```
public boolean isXoObjectWelcome(XoObject maybeChild, int index);
```

Informs whether XO object argument could be added as child of current container, from an implementation point of view. It's in never case an XML validity checking. This method is useful to avoid exception when attempting to add implementation-incompatible objects.
For instance, considering an XML element <panel> implemented by a class derived from javax.swing.JPanel ; if this container can only accept XO children derived from java.awt.Component, isXoObjectWelcome() is the right method to perform such a check.

```
public XoObject removeXoChild(int index) throws
ArrayIndexOutOfBoundsException;
```

Removes XO child at index argument from the children list. If index is invalid, that is, no child is at this index, an XoException must be thrown. The method returns removed object. Just before returning it, the method should set its parent property value to null, by calling setXoParent(null).

Note : addXoChild() and removeXoChild() can only be directly used by developer when building a tree which do not belong to an XoDMInstance object, at the condition that the child is not polymorphic (see Polymorphism section in this chapter). The class XoUtilities provides useful method to handle any kind of child. In any cases, to add or remove a child to or from a container belonging to an XoDMInstance tree, developer must call corresponding methods provided by XoDMInstance class.

2.3.2 Children access methods

```
public int getXoChildrenCount();
public XoObject[] getXoChildren();
public XoObject getXoChildren(int index) throws
 ArrayIndexOutOfBoundsException;
```

Here above are standard methods to access to children in a tree structure. Note that getXoChildren() must always return non null array : if there is no child, return empty XoObject[].

2.3.3 Listeners

```
public void addXoContainerListener(XoContainerListener l);
public void removeXoContainerListener(XoContainerListener l);
public XoContainerListener[] getXoContainerListeners();
```

An event, XoContainerEvent, is fired each time an XO object is added or removed as child of a container belonging to an XoDMInstance content tree. This is internal core system which manage firing of this event, developers don't have to care about. On the other hand, each XoContainer object must keep track of XoContainerEvent listeners, and must be able to provide the listener list on demand from internal system, in order for it to fire event when an add or remove method call is performed on XoDMInstance object.

2.3.4 Content-Model

```
public XoContentModel getXoContentModel();
```

Xotics API manages XML content-model as defined in XML schema standard. For an XoContainer class, XML content-model is a regular expression describing which kind of children are allowed and how they must be organised, from an XML point of view. That is, control is done with XML identity of children.
The content-model of a particular container class can be defined at two levels : in Data-Model definition document (DMDL) or/and by return value of this method, getXoContentModel(). Content-model defined in DMDL document is called "static content-model", whereas getXoContentModel() returns a "dynamic content-model". Indeed, in Xotics API, content-model is allowed to change depending on any application runtime condition. When content-model is just static, this method must return null. If non null XoContentModel value is returned, associated dynamic content-model takes priority on possible defined static content-model.
Static or dynamic, XML content-model is represented in Xotics environment as a tree of XoContentModel object nodes. This tree is a Java version of regular expression used to describe a content-model. Developer have to handle XoContentModel objects only when defining dynamic content-models.
Please refer to chapter 3.1 (DMDL dialect) for more details.

2.4 XoTextContainer interface

This kind of XO object is used to represent XML pattern below :

```
<element>text</element>
```

that is, an element having possible attributes, and especially whose content is PCDATA only. This PCDATA content is mapped into a particular XoTextContainer property, named xoTextContent, which can be of any type. XoTextContainer is the implementation of Simple Content Element concept defined in XML Schema standard.
This interface is a signature to inform Xotics environment that implementation class has a property named xoTextContent which stands for XML PCDATA content. Thus, this property must be defined in respect of JavaBean writing standard, having type of developer's choice. It's because PCDATA content can be mapped into a property of any type that XoTextContainer has no accessor for xoTextContent property.
XoTextContainer has two methods, as shown below, the accessors for a cdataSection property. This property tells whether PCDATA content must be written in XML with enclosing CDATA section characters.

```
public boolean isCdataSection();
public void setCdataSection(boolean cds) throws PropertyVetoException;
```

2.5 XoText interface

This interface represents a single text node, just a PCDATA content. When XML content-model of an XoContainer mixes text with elements, textual parts are mapped into XoText objects.
XoText classes are very similar to XoTextContainer, with the same xoTextContent property used to represent PCDATA, excepted that its type is fixed as String. When time has come to save an XO object tree into XML document, xoTextContent property value is written as PCDATA where XoText object stands in the tree.
XoTextSupport is the default implementation class, which can be (and is) used directly.
During parsing of an XML document to create XoDMInstance representation, when PCDATA is met, Xotics environment asks for current XoContainer whether it can accept an XoText object as child to hold PCDATA value, by calling its method isXoObjectWelcome(). If container accepts, environment uses either default or custom implementation (if one is defined in DMDL document). If container refuses, PCDATA is discarded.
Here are XoText interface methods, with same properties as XoTextContainer :

```
public boolean isCdataSection();
public void setCdataSection(boolean cds) throws PropertyVetoException;
public String getXoTextContent();
public void setXoTextContent(String pcd) throws PropertyVetoException;
```

2.6 XoRoot interface

XoContainer classes, usable as root element representations, must implement XoRoot interface. In a Data-Model, any number of XoRoot classes can be defined, including none. In this later case, no document can be created with this sole dialect implementation, another Data-Model containing XoRoot class(es) is needed as support dialect.

```
public void xoInitialize(Object initObject) throws XoException;
```

This method is called on root element of a newly created XoDMInstance object. Its implementation is optional, it can be used to perform some user defined init process on the document. Object argument can be provided at creation invocation to help at initialization. This object is specific to expected XoRoot class. The method can throw an XoException to inform about init failure.

```
public void xoRelease();
```

This method is called by XoDMInstance.release(), called itself to release allocated resources when XoDMInstance object is no more used. This method is also optional, it's the opposite of xoInitialize().

```
public void checkXoDMInstanceValidity() throws XoValidityException;
```

This method is the place to contain global validity checking code. Developer is free to include any specific code to check document validity. This method is called by XoDMInstance.checkValidity() with no argument, which means document wide checking.

2.7 Polymorphism

Polymorphism is a Xotics API feature by which one XML element can be represented by several different XO classes in a Data-Model, the choosen implementation depending on its place in a document. Polymorphism is the Xotics implementation of Local Element concept from XML Schema standard.
As an example, consider a dialect where an element <cell> can be found in content-models of <grid> and <table> elements. As child of <grid>, <cell> must be represented by an XO object derived from javax.swing.JPanel and, when <cell> is child of <table>, its associated class derives from XoObjectSupport.
Such constraints are easy to resolve, each XO object must implement XoPolymorph interface.
In a specific Data-Model, all XO classes representing a single XML element must implement XoPolymorph. This interface contains accessor methods to a special XO object called XoPolymorphWrapper, which holds all implementations of an XML Local Element.
With an XML local element, the implementation class to choose is found in the content-model of each possible parent element. When such content-model has a reference on the local element, associated content-model node must have the attribute/property "elementType" set to the "type" attribute value of appropriate XO implementation class (see chapter 3.1 DMDL dialect), in order to designate a specific implementation for the element.

2.7.1 XoPolymorphWrapper

With polymorphism mecanism, two states are to be considered : when a polymorph XO object (i.e implementing XoPolymorph) belongs to an XoDMInstance, and when it doesn't. These two states form a cycle as follows :
When it doesn't belong to a DM instance (aka document or XO object tree), a polymorph object is wrapped by a polymorph wrapper (of class XoPolymorphWrapper), which implements XoObject interface and thus can take the place of the polymorph. The wrapper knows which implementation can be suitable as child of a specific container. So, when the wrapper is being integrated in a document, a search is performed in the content-model of the new parent and the wrapper is replaced by appropriate XO class. This implementation object keeps a reference on its wrapper so, when it is removed from the document, the wrapper replaces the XO object again.
This mecanism is possible only when respecting some constraints :
First, an XO object implementing a local element must be created by a call to createXoObject() of the Registry, which is the only method able to create and setup a polymorph wrapper. Thus, giving a local element full name as argument, createXoObject() returns a polymorph wrapper containing all possible implementations for this element.
Then, adding and removing the representation of a local element must be performed by calls to the methods addChild() and remove() of XoUtilities or XoDMInstance.

Chapter 3 : Xotics Data-Model

To be usable in Xotics environment, every XML dialect implementation must be loaded, by the Registry. The process consists in providing to the Registry a DMDL formatted document, describing dialect implementation, in other words, Xotics Data-Model. This DMDL document can be provided alone or packaged in a JAR file containing all necessary classes. Then, the Registry can give all relevant informations about the Data-Model and can perform useful tasks.

3.1 DMDL dialect

DMDL (Data-Model Definition Language) is the definition format of any XML dialect implementation available in Xotics environment. DMDL document content can be classified into two categories :

- general purpose informations,
- informations specific to each XML element implementation.

3.1.1 General purpose informations

Below is the list of these informations, followed by detailed description of DMDL elements concerned :

- dialect namespace URI, vendor or provider and implementation version ID,
- classpaths, as URLs, to locate classes designated in the document,
- XML element list,
- default init options, for loading or creating XoDMInstance made with this Data-Model,
- generic PropertyEditor classes to use to convert and edit specifics Java types,
- new XPath functions and XSD datatypes proper to this Data-Model,
- validation rules to apply to any XoDMInstance made with this Data-Model.

3.1.1.1 Element <definition>

This is the root element of any DMDL document. It holds essentially the namespace URI, used in Xotics environment as unique ID of each loaded Data-Model. Its children elements are the containers of every information categories.

Note : all <definition> children are optional, even dialect element list ; indeed, a Xotics Data-Model can be loaded for its XPath functions or XSD datatypes only.

Content-Model :	
<definition> contains following elements, in any order, with respective occurrence of 1 max : <init-options>, <elements>, <xpath>, <validity-rules> et <editors>. Which can be expressed by this regular expression : (init-options? & elements? & xpath? &validity-rules? & editors?) note that «&» stands for operator ALL from XML Schema standard	
Attributes :	
refURI	required, represents namespace URI identifying this dialect implementation. Any String is valid, provided it is unique for each loaded Data-Model.
vendor	Provider name
version	implementation version

3.1.1.2 List Elements (<elements>, <init-options>, <editors>, <classpaths>, <validity-rules>)

These elements serve only to structure a DMDL document in distinct categories.

Content-Models :
<elements> contains a list of at least one <element>
<init-options> contains a list of at least one <init-option>
<editors> contains a list of at least one <editor>
<classpaths> contains a list of at least one <classpath>
<validity-rules> contains a list of at least one <validity-rule>

3.1.1.3 Element <xpath>

<xpath> holds a list of XPath language extensions, that is, functions and new XSD datatypes. For more information about extending XPath language, please refer to XPath chapter. These extensions are available to any XoDMInstance having a mapping for their namespace (equivalent to xmlns attribute). Read Data-Model Instance chapter, section namespace mapping for detailled informations.

Content-Model :	
A list containing at least one <function> or <datatype> : (function | datatype)+	
Attributes :	
prefix	required, prefix standing for namespace, used to identify functions and datatypes in XPath requests.

3.1.1.4 Elements <function> and <datatype>

Attributes :	
(for <function>) : functionClass (for datatype) : dtClass	required, Java full class name of XPath function or XSD datatype.

3.1.1.5 Element <init-option>

This element provides a default init option for any Data-Model instance created with a root element of current Data-Model. Default Init options can be overriden at XoDMInstance creation time.
There four different init options :

- READ_ONLY : makes a Data-Model instance to be set in read-only mode at creation time. Can have "true" or "false" value.
- REQUESTABLE : tells whether a DM instance can be XPath requested. When it is known that no XPath request will be done on DM instance made with current Data-Model, this option should be set to "false", in order to increase performance (memory and CPU).
- HISTORY_SIZE : set the count of historized modifications on DM instance. Value is any valid signed integer. Two values have special meaning : "-1" for unlimited size, "0" to deactivate historization.
- DMD_AUTO_LOADING_ENABLED : indicates whether Xotics environment is allowed to load "on the fly" appropriate Data-Models needed to complete XML document parsing.

Attributes :	
name	required, to choose among : READ_ONLY, REQUESTABLE, HISTORY_SIZE, DMD_AUTO_LOADING_ENABLED.
#PCDATA	depends on selected option name : READ_ONLY needs «true» or «false», REQUESTABLE waits for «true» or «false», HISTORY_SIZE needs integer value, -1 to specify "unlimited", 0 to tell "no history", DMD_AUTO_LOADING_ENABLED waits for «true» or «false».

3.1.1.6 Element <editor>

This element specifies a java.beans.PropertyEditor class to edit and convert a particular Java type. This is a generic association ; the PropertyEditor is used for each property encountered having associated Java type.

Attributes :	
targetClass	required, full class name of some property Java type.
editorClass	required, full name of a class implementing java.beans.PropertyEditor.

3.1.1.7 Element <classpath>

This element provides a URL as class path to any classes or resources specified in DMDL document, element implementations, property types, later loaded resources or anything else. When reading a DMDL document, Xotics environment first adds these URLs to its ClassLoader (XoClassLoader), in order to be able to load all classes mentionned. <classpath> is generally used when a Data-Model is deployed as a stand-alone DMDL document : the document can be placed at different location (on different servers for instance) than implementation classes.

Attributes :	
url	required, can locate a directory or JAR file.

3.1.1.8 Element <validity-rule>

This element defines a validity rule applied on any DM instance whose root element belongs to current Data-Model. The rule is an XPath request, whose result must be boolean, processed when calling XoDMInstance.checkValidity() with no argument. If result is false, validation fails.

Attributes :	
#PCDATA	a valid XPath request, as String

3.1.1.9 Element <element>

Describes an XML element. <element> holds in attributes general informations such as XML name or if it can be used as root element. It's essentially a container for implementation informations provided by <object> element. When there is more than one <object> child, each object described must implement XoPolymorph interface, since the element is implemented by several XO classes.

Content-Model :	
<element> has as much <object> children as there are implementations, at least one.	
Attributes :	
name	required, element local name
rootable	Tells whether this element can be found as root element (default is false)
displayName	String to display to designate this element in Xotics Editor application (null by default)
order	required, element number ID, must be unique in current Data-Model
hidden	Indicates whether this element can be displayed in Xotics Editor application (default is false)

3.1.2 Informations about element implementation

3.1.2.1 Element <object>

<object> and its descendant elements describe a single XML element implementation. This element essentially gives XO class name. Children describe :

- content-model, for container objects,
- published properties, that is, properties which represent attributes,
- JavaBean Customizer list to edit object.

For container object, specified content-model is a default content-model, called static, and can be overriden by XoContainer.getXoContentModel() method. No content-model defined means a content-model of type EMPTY, excepted of course if getXoContentModel() returns non null value.
To set which properties are published, one can let Xotics environment analyse XO object class to extract all valid properties, or specify which ones are published individually, or combine both possibilities. By default, if nothing is done, no property is published.
A list of several Customizers can be provided because Xotics Editor application can manage them all.

Content-Model :	
<object> contains in following order : an optional list of <customizer>, one optional <content-model> and also an optional list of <property>. (customizer*, content-model?, property*)	
Attributes :	
name	required, full Java class name of XO object implementation. This class must obviously implement at least XoObject interface, and also XoPolymorph when current <object> element has sibling <object> elements
type	String used to distinguish each XO object class in case of multi-implemented element (polymorphism)
default	when several implementations are defined, this boolean attribute sets default implementation, to choose when none seems to fit. Among <object> siblings, only one can have this attribute set to "true", "false" is default value.
idName	optional, the name of property serving as ID attribute/property (i.e whose value serves to index owner object in a DM instance).
beanScanning	when set at "true", tells Xotics environment to publish all valid JavaBean properties of XO class. Default is "false"

3.1.2.2 Element <customizer>

<customizer> binds a java.beans.Customizer class to XO object described.

Attributes :	
customizerClass	Required, full java class name implementing java.beans.Customizer.
displayName	optional, a name to identify this customizer in Xotics Editor application.

3.1.2.3 Element <property>

This element tells to publish or exclude from publishing a specific property of current XO class. Publication is the process by which Xotics environment is asked to consider a property as Java representation of an XML attribute.
if "beanScanning" attribute, on parent <object> element is set at "true", <property> can be used :

- to exclude a specific property from the list of published ones, by setting "exclude" attribute to "true",
- to give additional informations about specific property published, that is, a particular PropertyEditor class for this property only, and/or an XML attribute name different than default one, which is property's name.

if "beanScanning" is "false", each <property> element gives a property to publish, possible specific PropertyEditor class, and XML attribute name if it must be different of property's name.

Content-Model :	
<property> contains one optional <editor> element, which designate specific PropertyEditor class to edit and convert property's value.	
Attributes :	
name	required, JavaBean property name of XO object class
xmlAttrName	optional, sets XML attribute name associated to this property. By default (null), property and attribute names are identical.
exclude	when set at "true", tells to exclude this property, which would be published otherwise. "false" by default.

3.1.2.4 Element <editor>

When used as <property> child, this element binds a PropertyEditor class to the property to edit and convert its value. This association overrides possible generic binding done with <editor> element (as child of <editors>) on the same property type. "targetClass" attribute is not used in this context.

Attributes :	
editorClass	required, full name of a class implementing java.beans.PropertyEditor.

3.1.2.5 Element <content-model>

<content-model> describes XML content-model of current XO class, when class implements XoContainer. Content-model description format is a tree of <content-model> elements, very similar to Content-Model description of XML Schema standard. The tree is a "parsed" form of a regular expression, with values and operators, informing about which kind of elements are allowed as children of current container, and how they must be organised. To distinguish between them, <content-model> as an attribute "cmType" which defines each node type in the expression. cmType can have one of these values :

- CHILDREN : possible root type of <content-model> tree, indicating that children of container are exclusively elements (i.e with no text node);
- MIXED : another possible root type, indicating that children of container are elements or text nodes (that is objects implementing XoText) ;
- EMPTY : last possible root type and sole node, to inform that container can have no child ;
- CHOICE : represents operator «|» (choice) from DTD format ;
- SEQUENCE : represents operator «,» (sequence) from DTD format ;
- ALL : represents operator «&» (connector) used in XML Schema standard.
- ELEMENT_REF : reference on particular element ;
- ANY : reference on either any element (*:*), or every element of a particular namespace (namespace:*).

Content-Model :	
Children of <content-model> are also <content-model> elements. Its count depends on cmType attribute value. First, a <content-model> tree has a root whose cmType attribute is CHILDREN, MIXED or EMPTY. Then, depending on cmType, allowed children are very specific : CHILDREN or MIXED : <content-model> must have one single child, of any type excepted CHILDREN, MIXED and EMPTY. EMPTY : <content-model> has no child. CHOICE, SEQUENCE ou ALL : <content-model> accepts at least two children, of type CHOICE, SEQUENCE, ALL, ELEMENT_REF or ANY. ELEMENT_REF or ANY : <content-model> can't have any child.	
Attributes :	
cmType	required, type of <content-model> element
minOccurs	minimal occurrence count of a child element or group of elements, by default at 1, must be just inferior or equal to maxOccurs. Special value "UNBOUNDED" means unlimited count.
maxOccurs	maximal occurrence count of a child element or group of elements. By default at 1, must be superior or equal to minOccurs. Special value "UNBOUNDED" means unlimited count.
elementRef	local name of a referenced child element, required if cmType is ELEMENT_REF, unused otherwise
elementType	identifier used to designate a particular implementation class, in case of multi-implemented element. Required if referenced child element is implemented by several XO classes, optional otherwise
nsRef	dialect namespace URI, used when cmType is ELEMENT_REF or ANY. In first case, nsRef indicates the child element's namespace. In case of ANY, nsRef is used to restrict allowed children elements to those belonging to specified namespace. As for ELEMENT_REF, a null nsRef value means that namespace is the same as described Data-Model's namespace. As for ANY, null nsRef means that any child element, from any namespace, is allowed

Note : the content-model of <content-model> element is typically a good example of what can be a dynamical content-model. DMDL implementation itself, which is a core Xotics Data-Model, describes its content-model dynamically (since it depends notably on cmType attribute value), with a particular writing of getXoContentModel() method.

Here are some examples :
Content-model (frame | button)* is written in DMDL as follow :

```
<content-model> cmType =”CHILDREN”>
  <content-model> cmType=”CHOICE” maxOccurs=”UNBOUNDED” minOccurs=”0”>
    <content-model> cmType=”ELEMENT_REF” elementRef=”frame”/>
    <content-model> cmType=”ELEMENT_REF” elementRef=”button”/>
  </content-model>
</content-model>
```

Here is the format of (ANY) content-model :

```
<content-model> cmType =”CHILDREN”>
  <content-model> cmType=”ANY”/>
</content-model>
```

To represent a content-model defined as « a sequence of : 3 <frame> children no more, unlimited count of <button> children, then a single child belonging to namespace ns1», we would write :

```
<content-model> cmType =”CHILDREN”>
  <content-model> cmType=”SEQUENCE”>
    <content-model> cmType=”ELEMENT_REF” elementRef=”frame” maxOccurs=”3”
minOccurs=”0”/>
    <content-model> cmType=”ELEMENT_REF” elementRef=”button”
maxOccurs=”UNBOUNDED” minOccurs=”0”/>
    <content-model> cmType=”ANY” nsRef=”ns1”/>
  </content-model>
</content-model>
```

3.2 Data-Model deployment

A classical way to load a Xotics Data-Model is to provide DMDL document describing it. The sole constraint is then to make sure that referenced classes in the document can be effectively accessed, either by JVM classpath or with help of <classpath> elements in document.
The other way to load a Data-Model is to package both DMDL document and all necessary classes in a JAR file.
In order for JAR file to be valid, a special entry must be included in its manifest, Definition-Path, which gives access path to the DMDL document file. Here is a valid manifest sample :

```
Manifest-Version: 1.0
Created-By: Virtual Weaver Company
Definition-Path: com/virtualweaver/xotics/dummy/Dummy1Def.dmdl
```

3.3 The Registry

The main purpose of the Registry is to (un)load Xotics Data-Models, and make accessible its description. The Registry, of XoRegistry class, is instanciated once by XoEnvironment instance. It's accessed by getRegistry() method of XoEnvironment and XoDMInstance classes.

3.3.1 Data-Model loading

Here is (un)loading methods provided by XoRegistry class :

```
public String loadDataModel(XoDMInstance dm) throws XoException;
public String loadDataModel(URL jarUrl) throws XoException;
```

First method loads the Data-Model defined by DMDL document given as a DM instance argument, and returns the namespace URI identifying loaded Data-Model.
Second method loads the Data-Model packaged in a JAR file located by URL argument. It is expected that this JAR has a manifest conformant to Xotics requirements.

```
public boolean hasDataModel(String nsref);
```

The Registry identifies each Data-Model by its namespace URI. When loading a new Data-Model, if an already loaded Data-Model has the same namespace, it silently unload it to load the new one, excepted for core Data-Models, such as DMDL itself. To check or avoid this case, hasDataModel() can be called.

```
public void unloadDataModel(String nsref) throws XoException;
```

Unloading method.

3.3.2 Access to Data-Model informations

The Registry provides numerous access methods to Data-Model informations. The common part of all these methods is the following arguments :

- for general purpose informations, the namespace identifying Data-Model concerned,
- an XoObject instance, for getting informations about it.

```
public XoObject createXoObject(String nsref, String element) throws
XoException;
```

This frequently used method instanciates XO object associated to the element whose namespace and local name are given in arguments. This is not the only way to create an XO object, it can be instanciated by calling its constructor, providing that its xmlNameSpace and xmlLocalName properties are correctly setted and above all, that this XO object is the sole implementation of its bound XML element. Indeed, for element implemented by several XO classes, there is no way to know, at creation time, which XO object is to choose, as selection depends on where it will be added in a document.

This method offers a generic way to create any XO object, valid for every case. When element whose reference is given in arguments is multi-implemented, this method returns an XoPolymorphWrapper object containing all available implementations. Since XoPolymorphWrapper is an XO object, it can be added normally in a DM instance content ; at this moment, appropriate implementation takes place.

```
public XoDMInstance getDataModelDocument(String nsref) throws XoException;
public XoDMInstance[] getDataModelDocuments();
public URL getDMLocation(String nsref) throws XoException;
```

The two first methods returns DMDL documents, as DM instances, of either specific or all Data-Model loaded.
Third method returns the location URL of JAR file source, when one exists, or null.

```
public XoProperty[] getProperties(XoObject xo) throws XoException;
public int getPropertyCount(XoObject xo) throws XoException;
public XoProperty getProperty(XoObject xo, String pName) throws XoException;
public XoProperty getPropertyByXml(XoObject xo, String xmlName) throws
XoException;
```

Here is access methods to published properties of a specific XO object. Properties are wrapped by XoProperty objects. The last two methods get respectively a speicif property by its property name and XML attribute equivalent.

```
public String getType(XoObject xo) throws XoException;
public String getIdName(XoObject xo);
public XoCustomizerInfos[] getCustomizers(XoObject xo);
```

First method returns the implementation identifier (named type) of XO object argument, used to identify a specific implementation among several ones of a single XML element.
Second method returns the name of the property used as ID, when it exists, of XO object argument.
The third method retrieves the Bean Customizers used to edit XO object argument, in an array of couples (java.bean.Customizer class, customizer display name).

Chapter 4 : Data-Model instance

A Data-Model Instance is the image of an XML document in a Xotics Environment. This essential object concentrates numerous operations on XO object tree. Its Java class is XoDMInstance. On creation, a DM instance takes ownership of an XO object tree whose root is allways an XoRoot object. Such a tree can have only one DM instance owner, and root object can not change of owner until DM instance release.

4.1 The Factory

The purpose of the Factory is to create every Data-Model Instance, new blank one or created by loading an XML document. The Factory Java class is XoFactory. It's instanciated and provided by a Xotics Environment, accessible by the method getFactory() of an XoEnvironment instance.

4.1.1 Loading

```
public XoDMInstance loadDMInstance(URL url, Map options, Object initObject)
throws XoException;
public XoDMInstance loadDMInstance(Reader reader, Map options, Object
initObject) throws XoException;
public XoDMInstance loadDMInstance(InputStream is, Map options, Object
initObject) throws XoException;
public XoDMInstance loadDMInstance(String doc, Map options, Object
initObject) throws XoException;
```

An XML document loaded by loadDMInstance() is accessible by four different ways, by URL, a Reader or an InputStream, and also by a String containing the document.
This method can override default creation options, such as defined in root element's Data-Model, by putting them in options parameter. Last argument is an arbitrary object directly provided as parameter for xoInitialize() of the root element, called at last step of XoDMInstance instanciation. This object depends on root's Data-Model and so can be optional.
Loaded XML document must obey one constraint : each element must be identified by a namespace, with xmlns special attribute, in order for Xotics core system to find associated XO object implementation.

4.1.2 Creation

```
public XoDMInstance createDMInstance(String rootnsref, String rootnsprefix,
String rootelement, Map options, Object initObject) throws XoException;
public XoDMInstance createDMInstance(XoRoot root, String rootnsprefix, Map
options, Object initObject) throws XoException;
```

There is two ways to instanciate an XoDMInstance object which doesn't come from an existing XML document. In both cases, root element must be explicitly specified. This one is immutable in an XoDMInstance, it can neither be removed nor replaced.
Thus, createDMInstance() takes all parameters needed to choose a root element. The two last arguments, options and initObject are employed in the same conditions as for loadDMInstance().
In first method version, XoRoot object is created from XML element name and namespace. The other version creates a DM instance from an existing XoRoot object, single or root of an XO object tree, provided none of these objects already belongs to another valid DM instance.

4.2 Export

This XoDMInstance feature is another way to create a DM instance. It consists in cloning a part of tree content of an existing DM instance to integrate it in a new DM instance. The root of the cloned sub-tree must be, of course, an XoRoot object. It can be root of source DM instance or any descendant of type XoRoot.

```
public XoDMInstance exportDMInstance(XoRoot rpos) throws XoException;
```

Note : calling this method to export a DM instance from its root object is equivalent to a call to clone() method of XoDMInstance.

4.3 Saving

An XoDMInstance object is stored as XML textual document by calling save() method, which is declined in four versions :

```
public void save(URL url) throws IOException, XoException
public void save() throws IOException, XoException
public void save(Writer wr) throws IOException, XoException
public void save(OutputStream os) throws IOException, XoException
```

First version saves document at location defined by URL parameter. Currently, file and http protocols are supported. Second save() method, without argument, is similar to previous one, the URL being stored by following method :

```
public void setSourceLocation(URL loc) throws XoException
```

This method is called also internally in the following cases :

- at each call of save(URL) ;
- when creating an XoDMInstance object by loadDMInstance(URL, …) method of the Factory ;

The two last versions send XML document content to stream Writer or OutputStream.

A DM instance has a property storing charset encoding to use when saving. It is accessed by these methods :

```
public void setEncoding(String charsetEncoding) throws XoException;
public String getEncoding() throws XoException;
```

Note : when using save(Writer), make sure that the writer parameter is created with same charset encoding as DM instance one, got by getEncoding() method.

4.4 Release

An XO object tree can be used without belonging to any DM instance, but while belonging to a DM instance, programmer must use DM instance owner to handle the tree content. This tree can be made free by calling this method :

```
public void release() throws XoException;
```

All DM instance resources are released, including content tree, which can be reused for any purpose. When calling this method, xoRelease() of XoRoot is invoked. One can know whether an XoDMInstance is released by calling :

```
  public boolean isReleased();
```

4.5 Content handling

4.5.1 Add/Remove

A DM instance is a class embedding an XO object tree. Any modification of tree content structure must be done by appropriate XoDMInstance methods. Such modification consists in adding or removing an XO object to or from content tree. For it, one can use one of the three following methods :

```
public XoObject addChild(XoObject child, XoContainer parent) throws
XoException;
public XoObject addChild(XoObject child, XoContainer parent, int index)
throws XoException;
public XoObject remove(XoObject o) throws XoException;
```

Calling these methods is required, because add/remove process in a DM instance is not just calling corresponding XoContainer methods.
Moreover, about an add process, XoDMInstance class is in charge of :

- Resolving appropriate implementation for polymorph element,
- Indexing objects,
- Historizing the operation for undo function.

addChild() methods return XO object really added to DM instance. This returned object will be different from XO object passed as argument when it represents a multi-implemented element. Then, this is an object of type XoPolymorphWrapper, transformed in appropriate implementation.
Similarly, remove() method returns the object passed as argument if its the unique implementation of associated XML element, and an XoPolymorphWrapper object if the argument belongs to several implementations.
Before integrating an XO object to a DM instance, one can know if it can be integrated by calling the following method, which is only the synchronized version of XoContainer equivalent method :

```
public boolean isWelcome(XoContainer parent, XoObject child, int index)
throws XoException;
```

4.5.2 Modification events

DM instance reports each modification on tree structure and property changes, which can be listen via these methods below :

```
public void addPropertyChangeListener(PropertyChangeListener l);
public void removePropertyChangeListener(PropertyChangeListener l);
public void addXoContainerListener(XoContainerListener l);
public void removeXoContainerListener(XoContainerListener l);
```

When receiving such events, source object is not DM instance, but the real event source (a container or a property's owner).

4.5.3 History / Undo

Each modification on a DM instance content, i.e add, remove and property change, can be historized and so, can be undone by calling the following method :

```
public void undo() throws XoException;
```

This operation rolls back the last modification, at the condition that history size is different of 0. This size is specified by HISTORY_SIZE init option. A size of -1 means unlimited history size. This size can be known by invoking this method :

```
public int getHistorySize();
```

Each modification on a DM instance content which can be undone yields a specific event XoUndoableEvent. XoUndoableListener object can (un)register via these methods :

```
public void addXoUndoableListener(XoUndoableListener l);
public void removeXoUndoableListener(XoUndoableListener l);
```

4.5.4 Import

With XoDMInstance, tree content of a DM instance can be imported in another DM instance, by cloning it and adding it as child of a XoContainer. Cloned DM instance is not modified by this operation.

```
public void importDMInstance(XoDMInstance edm, XoContainer ppos) throws
XoException;
```

4.6 Object requesting

XoDMInstance class provides several ways for searching or requesting about its content. The main one is XPath requesting.

4.6.1 XPath request

```
public XoDataType request(String reqIdOrXPath, XoNode ctx, Map xpathVars)
throws XoException;
public XoNode[] requestXoNodes(String reqIdOrXPath, XoNode ctx, Map vars)
throws XoException;
public boolean isRequestable();
```

The base method is request(). It takes a String parameter containing the request in XPath language, a context node (or null to indicate tree root), and an optional map containing XPath variables, i.e. couples of variable names and values. XPath variables and method return are of type XoDataType. This is an empty interface indicating a type compatible with request processing. Context node, as an XoNode, can be the XoDMInstance on which the request is done, any XoProperty or XoObject belonging to the tree content.
Second method is an "helper" version of request() used when requesting XoObject or XoProperty objects. If XPath request can not result in XO objects or properties, the method returns an empty array.

Note : using these XPath methods is possible only if DM instance has been created with REQUESTABLE init option. Otherwise, an exception is thrown when attempting to call them. You can use the method isRequestable() to know if XPath requests are possible on a DM instance.

4.6.2 Searching by ID

Every XO object belonging to a DM instance can have an ID. These two methods below take benefit from this feature. The first one searches for an XO object by its ID , second one gives the list of registered IDs.

```
public XoObject getXoObjectById(String id) throws XoException;
public String[] getReferencedIds() throws XoException;
```

4.7 Namespace management

Here below are described methods for managing and requesting prefixes of namespace used in an XoDMInstance.
As XML document, an XoDMInstance can have elements from several namespaces and so, must give a short prefix for every namespace present. This can be done by the method below :

```
public void addNSMapping(String nsref, String nsprefix) throws XoException;
```

addNSMapping() associates a prefix string for namespace given as paramter. But this method does more : it loads also extra informations provided by corresponding Data-Model, particularly possible XPath data.
You are not forced to used this method when an XoObject is added to an XoDMInstance content tree, its namespace is checked and if new, a prefix is automatically associated, of the form "nsxx" where xx is a number incremented as needed. You can then use changeNSMapping() below to change prefix if needed.

```
public void changeNSMapping(String nsref, String nsprefix) throws
XoException;
```

Note : you must use addNSMapping() either when you want to control prefix value, or when you want to use XPath datatypes and functions included in a specific Data-Model. Otherwise, this method is not required.

Here are the requesting methods about namespace management :

```
public boolean hasNSMapping(String ns) throws XoException;
public String getNamespaceURI(String nsprefix) throws XoException;
public String getRootNamespaceURI() throws XoException;
public String[] getNamespaceURIs() throws XoException;
public String getNSPrefix(String nsref) throws XoException;
```

4.8 Data-Model location Management

Data-Model Automatic Loading is a feature by which Xotics API, when creating an xoDMInstance from an XML document, can load Data-Models automatically, as needed, from a specified location. A specific XML Processing Instruction can be inserted in the loaded document to inform the core system that a particular Data-Model must be loaded in order to create the XoDMInstance correctly and, if not already loaded, it can be found at a specific location. Here is an example of such Processing Instruction :

```
<? dmd http://www.xotics.org/dummy.jar ?>
```

If INIT_DMD_AUTO_LOADING_ENABLED init option is set to true when the Factory loads an XML document containing this Processing Instruction, the Data-Model dummy.jar will be loaded if needed, to find appropriate XoObject implementation of a document element.

When XoDMInstance saves its content, such Processing Instructions can be written in the document. First, XoDMInstance must be allowed to do so, via this method :

```
public void setInsertDMLocation(boolean enabled) throws XoException;
public boolean isInsertDMLocation() throws XoException;
```

Then, you can get and modify Data-Madel locations that can be written as Processing Instructions :

```
public URL[] getDMLocations() throws XoException;
public URL getDMLocation(String nsref) throws XoException;
public void setDMLocation(String nsref, URL url) throws XoException;
```

4.9 Access Management

An XoDMInstance access can be restricted by two ways :

- it can be set in a read-only mode,
- it can be locked by a specific thread.

4.9.1 Read-only mode

Here are read-only mode accessor methods :

```
public void setReadOnly(boolean rd) throws XoException;
public boolean isReadOnly() throws XoException;
```

When the read-only mode is set, any attempt to change either the tree structure or a property will result in an exception thrown :

- for a tree structure change (addChild() or remove()), an XoException is thrown, with a reason set to XoException.READ_ONLY,
- for a property change attempt, a PropertyVetoException is thrown. This feature can work properly only for XoObject having implemented Property Veto feature from JavaBeans specs.

4.9.2 Locking

This feature gives the ability to restrict a DM instance access to a specific thread.

```
public void getLock();
public void unlock() throws XoException;
public boolean isLockOwner();
public boolean isLocked();
```

getLock() method waits until current calling thread gets exclusive access on current DM instance. Then, any call to isLocked() returns true, until the lock owner thread calls unlock() to release DM instance. When a lock is active on a DM instance, most XoDMInstance methods throw an XoException (with

reason set to XoException.DMI_LOCKED) if they are invoked by another thread than the lock owner. A thread can know if it is the lock owner by calling isLockOwner().

4.10 Validation checking

```
public void checkValidity() throws XoException;
public void checkValidity(XoObject o, boolean deep) throws XoException;
```

The first method checks all the document, whereas second method checks only an XoObject in the tree, alone or with its sub tree, depending on deep parameter value.
Validity checking is performed following this process :
First, for both checking methods, each object in the tree is checked in document order, by calling XoObject's method checkXoValidity(), then checking content-model validity, if current object is a container. To get content-model, Xotics core system asks for XoContainer's method getXoContentModel(). If the method returns null, the content-model is asked from Data-Model static description (at this point, a null value should be considered as EMPTY contenty-model).
Then the first checkValidity() method (i.e whole document validity checking) executes possible validity rules from Data-Model description, and root element's checkXoDMInstanceValidity().
About checkValidity() applied to an XoObject, whatever deep param value, content-model checking will be performed if o parameter is an XoContainer.

Chapter 5 : XPath support

A major feature of Xotics API is its XPath 2 support. With Xotics, it's possible to perform an XPath request on a DM instance. As XPath 2 uses XML Schema Datatypes (XSD), Xotics API provides an implementation for these datatypes, and new XS datatypes can be created by restriction, by enumeration or list, as described in XML Schema standard. Xotics API also provides a way to create new XPath functions.

5.1 XS Atomic datatypes

Xotics provides a specific Java class for each XS atomic datatype used by XPath 2 language. The main purpose of such a class is to hold a java value in a java type suitable to corresponding XS datatype. For example, Xotics implementation for XSD string holds a value of type java.lang.String. So, why wrapping a String in a specific class ? For several reasons :

- a Xotics XSD implementation supports a major feature of XSD standard : derivation by restriction,
- a Xotics XSD object is self converted into String and created from String so, no need to use a PropertyEditor,
- a Xotics XSD object is always valid (i.e it's impossible to create such object with an invalid value and held value is immutable).

5.1.1 XsdDataType class

All XSD java classes are in the package :

```
com.virtualweaver.xotics.datamodel.datatype
```

and class names start with prefix "Xsd".
All XS datatype classes implement XsdDatatype interface, which provides methods below :

```
public Class getJavaTypeClass();
public Object getJavaTypeValue();
public String getLocalName();
public String getNameSpace();
```

The first two methods informs about java type and value of current datatype. Notice there is no setter for value, because an XSD object is immutable : once created, its content value can not be changed.
Every XS datatype is identified by a namespace and a local name, which is given by the last two methods of XsdDataType.

Here below is a table of associated Java class for each atomic datatype :

Atomic Datatype	Xotics class name	java class of held value
anyURI	XsdAnyURI	java.net.URI
base64Binary	XsdBase64Binary	byte[]
boolean	XsdBoolean	java.lang.Boolean
byte	XsdByte	java.lang.Byte
date	XsdDate	java.util.GregorianCalendar
dateTime	XsdDateTime	java.util.GregorianCalendar
decimal	XsdDecimal	java.math.BigDecimal
double	XsdDouble	java.lang.Double
duration	XsdDuration	com.virtualweaver.xotics.datamodel.datatype.XdtDuration
float	XsdFloat	java.lang.Float
gDay	XsdgDay	java.util.GregorianCalendar
gMonth	XsdgMonth	java.util.GregorianCalendar
gMonthDay	XsdgMonthDay	java.util.GregorianCalendar
gYear	XsdgYear	java.util.GregorianCalendar
gYearMonth	XsdgYearMonth	java.util.GregorianCalendar
hexBinary	XsdHexBinary	byte[]
ID	XsdId	java.lang.String
IDREF	XsdIdRef	java.lang.String
int	XsdInt	java.lang.Integer
integer	XsdInteger	java.math.BigInteger
language	XsdLanguage	java.util.locale
long	XsdLong	java.lang.Long
Name	XsdName	java.lang.String
NCName	XsdNCName	java.lang.String
negativeInteger	XsdNegativeInteger	java.math.BigInteger
NMToken	XsdNMToken	java.lang.String
nonNegativeInteger	XsdNonNegativeInteger	java.math.BigInteger
nonPositiveInteger	XsdNonPositiveInteger	java.math.BigInteger
normalizedString	XsdNormalizedString	java.lang.String
positiveInteger	XsdPositiveInteger	java.math.BigInteger
QName	XsdQName	java.lang.String
short	XsdShort	java.lang.Short
string	XsdString	java.lang.String
time	XsdTime	java.util.GregorianCalendar
token	XsdToken	java.lang.String
unsignedByte	XsdUnsignedByte	java.math.BigInteger
unsignedInt	XsdUnsignedInt	java.math.BigInteger
unsignedLong	XsdUnsignedLong	java.math.BigInteger
unsignedShort	XsdUnsignedShort	java.math.BigInteger

Table 3 : implemented atomic XS datatypes

5.1.2 XsdAnySimpleType class

All atomic XSD classes derive from XsdAnySimpleType abstract class, which implements XsdDatatype, and contains methods to apply XSD derivation by restriction. They have a constructor of type

<init>(String) and a method toString() so, no PropertyEditor is needed to convert their value from or to XML text. The only way to set value is at creation time with constructor argument.

Atomic datatypes can be derived by restriction, by setting specific value for one or several XSD Facets. XsdAnySimpleType has the following static methods, which represent all possible facets for a datatype :

```
public static int getFractionDigit();
public static int getTotalDigit();
public static Object getMaxExclusive();
public static Object getMaxInclusive();
public static Object getMinExclusive();
public static Object getMinInclusive();
public static int getLength();
public static int getMaxLength();
public static int getMinLength();
public static Pattern getPattern();
public static byte getWhiteSpace();
```

Creating a new XS datatype by facet restriction consists first in overriding these methods to make them return the value of the new restrictions.
In XsdAnySimpleType, each method returns a default value, meaning that there is no restriction on the facet :

XSD Facet	Default value
FractionDigit	-1
TotalDigit	-1
MaxExclusive	null
MaxInclusive	null
MinExclusive	null
MaxExclusive	null
Length	-1
MaxLength	-1
MinLength	-1
Pattern	null
WhiteSpace	XoConstants.WS_PRESERVE

Table 4 : Default values for XSD Facets

The Exclusive and Inclusive getters return an object whose type depends on the Java class of the datatype value. In general, this is the same type as value's type, excepted for these datatype implementations, concerning "Exclusive" facet :

- XsdDouble.getMax/MinExclusive() return a java.math.BigDecimal, to cover the value Double.MAX_VALUE + Double.MIN_VALUE,
- XsdFloat.getMax/MinExclusive() return a java.lang.Double, to cover the value Float.MAX_VALUE + Float.MIN_VALUE,
- XsdLong.getMax/MinExclusive() return a java.math.BigInteger, to cover the value Long.MAX_VALUE+1,
- XsdInt.getMax/MinExclusive() return a java.lang.Long, to cover the value Integer.MAX_VALUE + 1,
- XsdShort.getMax/MinExclusive() return a java.lang.Integer, to cover the value Short.MAX_VALUE + 1,
- XsdByte.getMax/MinExclusive() return a java.lang.Short, to cover the value Byte.MAX_VALUE + 1,

The Pattern facet returns an object of type java.util.regex.Pattern, and WhiteSpace facet can return one of the following constant values, from XoConstants :

XoConstants.WS_PRESERVE : which means to keep XML value as it is,
XoConstants.WS_COLLAPSE : which collapse white spaces,
XoConstants.WS_REPLACE : which replace any white space (\n \r \t) by the Space char.

XsdAnySimpleType provides also the methods to test the matching of a value passed in constructor with every facet restriction :

```
protected void matchFractionDigit(int fd) throws IllegalArgumentException;
protected void matchTotalDigit(int td) throws IllegalArgumentException;
protected void matchMaxInclusive(Object o) throws IllegalArgumentException;
protected void matchMinInclusive(Object o) throws IllegalArgumentException;
protected void matchMaxExclusive(Object o) throws IllegalArgumentException;
protected void matchMinExclusive(Object o) throws IllegalArgumentException;
protected void matchLength(int len) throws IllegalArgumentException;
protected void matchMinLength(int len) throws IllegalArgumentException;
protected void matchMaxLength(int len) throws IllegalArgumentException;
protected void matchPattern(Pattern pt) throws IllegalArgumentException;
```

Each method test internal XSD value to check whether internal XSD value is conformant to parameter, depending on a specific facet. These methods can be called to add new restriction in an new XS datatype, as explain below.

Note : there is no mention of Enumeration facet of XSD standard. In Xotics API, this feature is implemented by another mecanism, a specific class XdtEnum, which is treated in § Enumeration.

5.1.3 Derivation by facet restriction

To create a new XS datatype, derived from an existing one with new facet restriction, you must follow the steps below. To help, we use the source code of XsdNonNegativeInteger. Indeed, XsdNonNegativeInteger has been created by restriction, derived from XsdInteger, so :

```
public class XsdNonNegativeInteger extends XsdInteger {
```

A nonNegativeInteger is an integer having a restriction of "0" on MinInclusive facet. To specify this new restriction, we must rewrite the static method getMinInclusive() as below :

```
public static Object getMinInclusive() {
  return new BigInteger("0");
}
```

At this point, we just have to add MinInclusive checking, using corresponding "match" method, giving the getMinInclusive() value as parameter :

```
public XsdNonNegativeInteger(String strv) throws IllegalArgumentException {
  super(strv);
  matchMinInclusive(getMinInclusive());
}
```

```
public XsdNonNegativeInteger(BigInteger bi) throws IllegalArgumentException
{
  super(bi);
  matchMinInclusive(getMinInclusive());
}
```

There is two constructors, but only the first one is required with XS datatype implementations, the other one is just a facility. Thus, a standard XSD integer is created, then this integer is checked to know if its value matches the MinInclusive facet with getMinInclusive() value.

We end the writing of the new datatype with redefinition of clone() and getLocalName() methods. For nonNegativeInteger, namespace remains the same as XSD integer.

```
public static String NAME = "nonNegativeInteger";

public String getLocalName() {
  return this.NAME;
}

public Object clone() {
  return new XsdNonNegativeInteger((BigInteger)this.value);
}

}
```

As you can see, internal value is accessible by protected field "value".

5.2 Enumeration

In XML Schema standard, Enumeration is a facet used to create new datatypes, exactly like the other facets. However, in Xotics API context, Enumeration facet is treated differently. Xotics API provides a class XdtEnum, in same package as XSD implementations, which can be used to implement XSD derivation by Enumeration facet, and more generally to implement any enumerated datatype.
Xotics API brings to forms of enumeration : standard and partial enumerations. The standard form defines a datatype whose value belongs to a finite set of possible constant values. The partial form defines a datatype whose value can be either choosen from a set of possible constants or any other value of a predefined datatype.
New enumerated datatypes, standard or partial, are created by deriving XdtEnum class, which is abstract.

There is two major benefits in using XdtEnum :

- XPath engine automatically recognizes it when performing request, and replaces it by its content value,
- Xotics environment associates, by default, a specific and graphical PropertyEditor to any property of class derived from XdtEnum.

5.2.1 XdtEnum class

This class is used to hold a value of a specific type, possibly choosen from a predefined set of constant values. It provides some code used by XPath engine and special PropertyEditors, and a framework to create enumerated types easely.

Here are the constructors :

```
protected XdtEnum(boolean ispart, XdtBidiMap v2s, String strValue) throws
IllegalArgumentException;
protected XdtEnum(boolean ispart, Object value, XdtBidiMap v2s);
```

As for the XS datatypes defined in Xotics API, an XdtEnum datatype sets its content value in constructor and nowhere else. So, there is two ways to create an enum, from a string form of the value or from the value itself. This value is respectively set by strValue or value params.
Excepted value setting, both constructors need the same parameters : a boolean indicating whether this enum is partial (i.e can accept another value than predefined constants), and a bidirectionnal HashMap, of type XdtBidiMap, containing the mapping between a value and its string form.
XdtBidiMap is a general purpose class located in datatype package of Xotics API.

Here below are the methods which must be redefined when creating any enumerated type :

```
public abstract Class getValueClass();
public static XdtEnum[] getEnumSpace();
```

An XdtEnum object must give the class of its value. This class is constant for a given XdtEnum derived class. getEnumSpace() gives the list of enumeration values, each one wrapped in a static final XdtEnum object.

Here below are the particular methods to redefine only when creating a partial enumeration :

```
protected Object createValue(String str) throws IllegalArgumentException;
protected String getAsString();
public static boolean isPartEnum();
public static Class getContentPropertyEditorClass();
```

Since we create a partial enum, unknown values can be provided, so methods must be written to create a value from a string and convert it as a string (a standard enum just need to provide a list of constant values with respective string form).
isPartEnum() has been added to give this information without need to instanciate.
The last method is optionnal : it must be overriden only if a specific PropertyEditor is needed to handle unknown value.

5.2.2 Enum datatype creation

As example, let's watch the source code of a class called XodAttributeUse (extract from Xotics Factory project), which is an enumerated type having some constant values as bytes. The code explained below gives a good idea of how to write a classical enumerated type derived from XdtEnum.

Let's start by giving the list of byte constants :

```
public class XodAttrUse extends XdtEnum {

  public static final byte OPTIONAL_USE = 0;
  public static final byte REQUIRED_USE = 1;
  public static final byte FIXED_USE = 2;
  public static final byte WITH_DEFAULT_USE = 3;
```

Then we must create the mapping between object values and their string forms, by using an XdtBidiMap object. This instance is static because these mappings are defined at class level :

```
  private static final XdtBidiMap vtos = new XdtBidiMap();
  static {
    vtos.putForward(new Byte(OPTIONAL_USE), "OPTIONAL");
    vtos.putForward(new Byte(REQUIRED_USE), "REQUIRED");
    vtos.putForward(new Byte(WITH_DEFAULT_USE), "WITH_DEFAULT");
    vtos.putForward(new Byte(FIXED_USE), "FIXED");
  }
```

We choose to create two constructors, the required one with a string as argument, and a useful one with a byte value :

```
  public XodAttrUse(String str) throws IllegalArgumentException {
    super(false, vtos, str);
  }

  private XodAttrUse(byte v) {
    super(false, new Byte(v), vtos);
  }
```

So, we can create static enum constants defining the enum value space :

```
public static final XodAttrUse OPTIONAL = new XodAttrUse("OPTIONAL");
public static final XodAttrUse REQUIRED = new XodAttrUse("REQUIRED");
public static final XodAttrUse FIXED = new XodAttrUse("FIXED");
public static final XodAttrUse WITH_DEFAULT = new
XodAttrUse("WITH_DEFAULT");

private static final XdtEnum[] enumSpace = new XdtEnum[] {OPTIONAL,
REQUIRED, FIXED, WITH_DEFAULT};

  public static XdtEnum[] getEnumSpace() {
    return enumSpace;
  }
```

And the last needed method, giving the java type of enum values :

```
  public Class getValueClass() {
    return Byte.TYPE;
  }
```

We have also redefined the clone() method and provided a method to get the value as byte :

```
  public byte getValue() {
    return ((Byte)getValueAsObject()).byteValue();
  }

  public Object clone() {
    return new XodAttrUse(getValue());
  }
```

5.3 XPath functions

As XPath 2 compliant, Xotics API enables to create new XPath functions to extend the set of base functions. An XPath function is a class implementing XpFunction interface from package xpath. Generally, an XPath function is created by derivation from XpfAbstractFunction, an abstract class implementing XpFunction and containing common base code :

```
public abstract class XpfAbstractFunction implements XpFunction {

public XoDataType evaluate(XpContext ectx, XpEvaluable[] args) throws
XoException;

public XoDataType evaluate(XpContext ectx, XoDataType[] args) throws
XoException;

public abstract String getLocalName();
public abstract String getNameSpace();

}
```

The original evaluate() method is the first version. This method uses the runtime context and the list of arguments to perform an operation returning an XoDataType. An argument is an XpEvaluable object, which produces an XoDataType. This method is written to evaluate all arguments then to call the second version of evaluate() with resolved arguments. So generally, first evaluate() method should be let as is and second version should be overriden to perform specific operation. You can override the first evaluate() method if you want to control the evaluation of XpEvaluable arguments.

Chapter 6 : Property Editor

Conversion between XML attribute and JavaBean Property is an essential task for Xotics API. As seen in section dedicated to properties (§ 2.2), a JavaBean Property Editor can be used to perform conversion between Java value and XML text representation of a Property unable to convert itself. Property Editor is also used in Xotics Editor to provide a visual editing of every XO object property.

6.1 Implementation basics

Xotics Environment uses only the following methods of a JavaBean PropertyEditor :

```
public String getAsText();
public setAsText(String text);
public Object getValue();
public setValue(Object value);
```

The Xotics Editor uses the following methods when editing an XO object property :

```
public boolean supportsCustomEditor();
public Component getCustomEditor();
public boolean isPaintable();
public void paintValue(Graphics g, Rectangle box);
```

For performance and (sometimes) compatibility reasons, if the PropertyEditor is designed to provide a user interface for edition, don't instanciate any graphical component until the method getCustomEditor() is called. So, the conversion methods should not use any graphical component to store value. It can also avoid some problems when running on OS without graphical system.

Xotics Core API provides an extension interface to PropertyEditor, XoPropertyEditor. This interface brings just one additional method :

```
public void setProperty(XoProperty p);
```

Xotics Environment automatically recognizes when a PropertyEditor implements this interface, and then sets the wrapper of the converted/edited property as soon as possible. Setting the property wrapper gives access to the property infos, the XO object owner and its DM instance, which can be useful. However, take care about the fact that such XoPropertyEditor can be used for conversion without property wrapper set.

Xotics Core API provides also several PropertyEditor implementations accessible in package com.virtualweaver.xotics.datamodel.editor.

6.2 XSD wrapper PropertyEditor

XML Schema Datatype implementation classes in Xotics Core API are self-convertible to and from text. They are also automatically recognized by Xotics Environment, which affect them a specific PropertyEditor classes (found in editor sub-package) useful for visual editing in Xotics Editor.
Using an XSD implementation class to represent an XSD type is very simple and interesting, but only possible when there is no Java type constraint : a dialect implementation can require existing Javabean with existing properties to represent some XML element and its attributes.

This is why Xotics Core API provides specific PropertyEditor classes called XSD wrappers, which convert and visually edit basic Java types as if they were XSD implementation classes. Not any Java type can be edited with those wrappers : it must be one of the Java types enclosed by XSD classes. Here below is a table of Java types which can be wrapped and corresponding PropertyEditor(s). All wrappers are in the package com.virtualweaver.xotics.datamodel.editor.wrapper.

Java type or derived from	Usable XSD wrapper PropertyEditor
java.lang.Boolean or boolean	XoXsdwBooleanPropertyEditor
java.lang.Byte or byte	XoXsdwBytePropertyEditor
java.util.GregorianCalendar	XoXsdwDatePropertyEditor XoXsdwDateTimePropertyEditor XoXsdwgDayPropertyEditor XoXsdwgMonthDayPropertyEditor XoXsdwgMonthPropertyEditor XoXsdwgYearMonthPropertyEditor XoXsdwgYearPropertyEditor XoXsdwTimePropertyEditor
java.math.BigDecimal	XoXsdwDecimalPropertyEditor
java.lang.Double or double	XoXsdwDoublePropertyEditor
java.lang.Float or float	XoXsdwFloatPropertyEditor
java.lang.String	XoXsdwIdPropertyEditor XoXsdwIdRefPropertyEditor XoXsdwNamePropertyEditor XoXsdwNCNamePropertyEditor XoXsdwNStringPropertyEditor XoXsdwQNamePropertyEditor XoXsdwStringPropertyEditor XoXsdwNMTokenPropertyEditor XoXsdwTokenPropertyEditor
java.util.Locale	XoXsdwLanguagePropertyEditor
Java.math.BigInteger	XoXsdwIntegerPropertyEditor XoXsdwNIntegerPropertyEditor (stands for negativeInteger) XoXsdwNNIntegerPropertyEditor (stands for nonNegativeInteger) XoXsdwNPIntegerPropertyEditor (stands for nonPositiveInteger) XoXsdwPIntegerPropertyEditor (stands for positiveInteger) XoXsdwUBytePropertyEditor (stands for unsignedByte) XoXsdwUIntPropertyEditor (stands for unsignedInt) XoXsdwULongPropertyEditor (stands for unsignedLong) XoXsdwUShortPropertyEditor (stands for unsignedShort)
java.lang.Integer or int	XoXsdwIntPropertyEditor
java.lang.Long or long	XoXsdwLongPropertyEditor
java.lang.Short or short	XoXsdwShortPropertyEditor

Table 5 : XSD wrapper property editors for java types

Internally, each wrapper uses corresponding XS datatype implementation class to perform conversion. So, when creating a new XSD type derived by restriction, a equivalent wrapper PropertyEditor can be created by just writing these lines :

```
public class MyRestrictedPropertyEditor extends XoXsdwIntegerPropertyEditor
{

  public MyRestrictedPropertyEditor() {
    super();
    xsdClass = MyRestrictedInteger.class;
  }
}
```

Here, MyRestrictedInteger class is a restriction from XsdInteger so, this PropertyEditor derives from XoXsdwIntegerPropertyEditor to edit an integer value. The field xsdClass is protected and must be set to the XSD implementation class used to perform conversion, which must derive from XsdAnySimpleType.

Chapter 7 : editing

7.1 Basics

In Xotics Editor, any XML document can be displayed by a custom panel called "document renderer". A renderer is responsible for providing :

- a custom view of a document, a view which can be specific to some particular XML dialect,
- Drag and Drop editing support,
- a user interface for any additional and specific editing/rendering features.

The Editing Extension of Xotics Core API provides fundamental classes to create and integrate a renderer into Xotics Editor.

The implementation of a document renderer is very permissive ; the only requirements are :

- the renderer class must derive from a Swing Container and must implement the interface XoDMRenderer, from package com.virtualweaver.xotics.editing.renderer ;
- if the renderer can select some node (element or document node itself), it must fire a particular event to inform the Editor of the selected node.

This selection event is XoDMEditEvent from package com.virtualweaver.xotics.editing.event. It contains the node selected, which can be either an XoObject or an XoDMInstance object. Corresponding listener is called on a unique method selected().

7.2 Rendering XML document

Here below is explained how to implement XoDMRenderer methods to create a valid renderer.

```
public void setDMInstance(XoDMInstance dm);
public XoDMInstance getDMInstance();
```

These methods are accessors of DM instance currently rendered. When setter method is called, the renderer must release the previously rendered document (if any) and perform rendering of the new DM instance.

```
public void addXoDMEditListener(XoDMEditListener l);
public void removeXoDMEditListener(XoDMEditListener l);
```

These methods are used to manage the list of XoDMEditEvent listeners. Implementing these methods is required only if this renderer wants to inform the Editor that some node is selected via its interface.

```
public XoNode getSelected();
```

This method must return at any time the currently selected node, or null if there is no currently selected node.

```
public void select(XoNode xo);
public void selectOver(int x, int y);
```

These are the two selection methods Xotics Editor can call on this renderer to force a particular selection. This selection must not fire any XoDMEdit event, as Xotics Editor doesn't need to be informed since these methods are called by it. Notice that about second method, (x, y) are expressed in renderer's coordinate system. If the selection can not be achieved, calling getSelected() must return null.

```
public void addDropTargetListener(DropTargetListener dtl) throws
TooManyListenersException;
public void removeDropTargetListener(DropTargetListener dtl);
```

As Xotics Editor supports Drag and Drop, it must be informed when an object is being dropped on this renderer, by using these registering methods. Notice that Xotics Editor adds one listener only. If the renderer can accept a drop operation on itself, it must implement these methods and getSelected() to provide the selected node when the drop operation is occurring. Typically, the renderer should have some code like this :

```
private DropTarget dropTarget;

...

dropTarget = new DropTarget();
dropTarget.setComponent(this);
```

dropTarget can be initialized in the constructor for instance. The method addDropTargetListener() could be written as follows :

```
public void addDropTargetListener(DropTargetListener dtl) throws
TooManyListenersException {
  dropTarget.addDropTargetListener(dtl);
}
```

If the renderer can initiate a drag operation, it must use one of the two Transferable described below to transfer data to drag, depending on the kind of data, which represent an XML element, instanciated or not.
If element to drag is an existing element of rendered document (notice that only an XoObject can be dragged, not a DM instance), please use XoObjectSelection from editing.util sub-package. It contains an XoObject. This drag operation is a move or copy operation.
If element to drag doesn't exist, it can be specified by a couple (namespace, local name) using XoElementRefSelection (from sub-package editing.util), which transfer an XoElementRef (from same package) object representing a fully-qualified XML element name. This drag operation corresponds to adding a new XO object.