OpenMI

The OpenMI Document Series

Part E - org.OpenMI.DevelopmentSupport technical documentation

For OpenMI (Version 1.4)

Title	OpenMI Document Series: Part E - org.OpenMI.DevelopmentSupport technical documentation for the OpenMI (version 1.4)
Editor	Peter Gijsbers, WL | Delft Hydraulics, Delft, The Netherlands
Authors	Rob Brinkman, WL | Delft Hydraulics, Delft, The Netherlands Peter Gijsbers, WL | Delft Hydraulics, Delft, The Netherlands
Document production	Peter Gijsbers, WL | Delft Hydraulics, Delft, The Netherlands
Current version	V1.4
Date	21/05/2007
Status	Final © The OpenMI Association
Copyright	All methodologies, ideas and proposals in this document are the copyright of the OpenMI Association. These methodologies, ideas and proposals may not be used to change or improve the specification of any project to which this document relates, to modify an existing project or to initiate a new project, without first obtaining written approval from the OpenMI Association who own the particular methodologies, ideas and proposals involved.
Acknowledgement	This document has been produced as part of the OpenMI-Life project. The OpenMI-Life project is supported by the european Commission under the Life Programme and contributing to the implementation of the thematic component LIFE-Environment under the policy area "Sustainable management of ground water and surface water managment" Contract no : LIFE06 ENV/UK/000409. The first version of this document has been produced as part of the HarmonIT project; a research project supported by the European Commission under the Fifth Framework Programme and contributing to the implementation of the Key Action “Sustainable Management and Quality of Water” within the Energy, Environment and Sustainable Development. Contract no: EVK1-CT-2001-00090.

Preface

OpenMI stands for Open Modeling Interface and aims to deliver a standardized way of linking of environmental related models. This document describes supporting classes as being utilized in the OpenMI Software Development Kit. It is the fifth document in the OpenMI report series, which specifies the OpenMI interface standard, provides guidelines on its use and describes software facilities for migrating, setting up and running linked models.

Other titles in the series include:

A. Scope

B. Guidelines

C. org.OpenMI.Standard interface specification

D. org.OpenMI.Backbone technical documentation

E. org.OpenMI Development Support technical documentation (this document)

F. org.OpenMI.Utilities technical documentation

The interface specification is intended primarily for developers. For a more general overview of the OpenMI, see Part A (Scope).

The official reference to this document is:

OpenMI Association (2007) *The org.OpenMI.DevelopmentSupport technical documentation*. Part E of the OpenMI Document Series

Disclaimer

The information in this document is made available on the condition that the user accepts responsibility for checking that it is correct and that it is fit for the purpose to which it is applied.

The OpenMI Association will not accept any responsibility for damage arising from actions based upon the information in this document.

Further information

Further information on the OpenMI Association and the Open Modelling Interface can be found on http://www.OpenMI.org.

Table of contents

1 Introduction

1.1 Background

OpenMI stands for Open Modeling Interfaces. It aims to deliver a standardized way to link environmental related computational models that run simultaneously. In summary, OpenMI primarily focuses on providing a complete protocol to explicitly define, describe and transfer (numerical) data between components on a time basis, including associated component access. It thus enables process interaction being represented more accurately, compared to sequential linkages.

The establishment of OpenMI will support and assist the scientific, consultancy and water management community in the integrated assessment of water management systems and thus strategic planning and integrated catchment management required by the European Water Framework Directive.

This standardized way of linking models is achieved by an intelligent protocol to describe, define and transfer data. This protocol is translated into a strict set of rules, i.e. formal interfaces, to be implemented by software code. Any component that implements these interfaces is called an OpenMI compliant component.

Within OpenMI, a distinction is made between the standardized interfaces, incorporated in the org.OpenMI.Standard namespace, and an implementation in other namespaces which provide a so-called Software Development Kit (SDK), see Figure 1. The org.OpenMI.DevelopmentSupport namespace provides generic support for configurations of linked OpenMI components. It contains a customizable generic XML-parser to store Compositions, LinkableComponent-OMI files and associated information on disk. The org.OpenMI.DevelopmentSupport package does not depend on any other packages within the org.OpenMI domain. Application of this software layer is not mandatory, but it may save costs and effort to join.

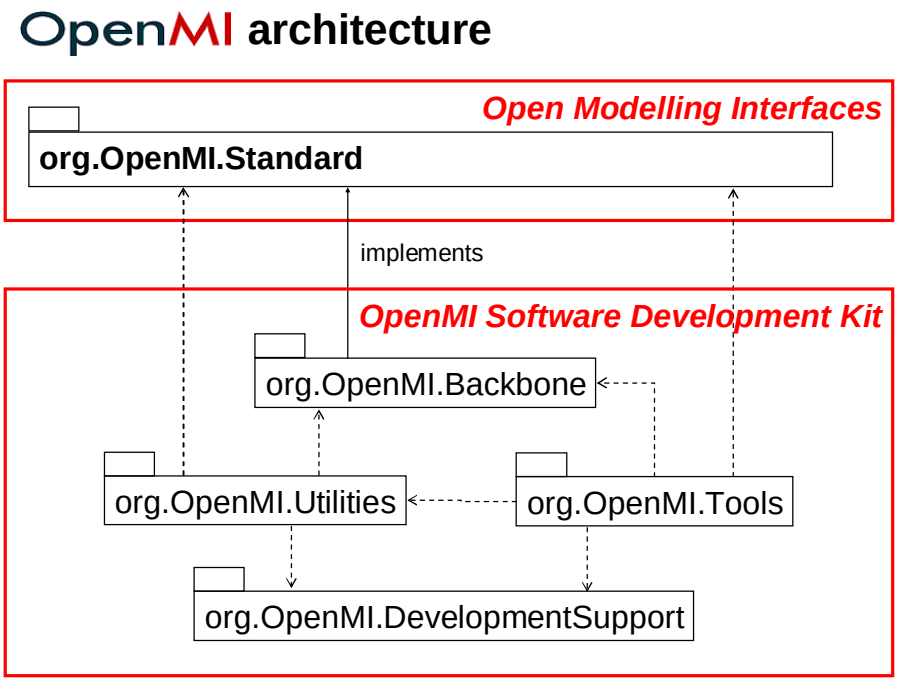

Figure 1 Namespaces in the OpenMI architecture

1.2 Requirements

The org.OpenMI.DevelopmentSupport namespace provides low level support to the implementation of the OpenMI architecture. The following requirements have been kept in mind.

- [Req-DS1] – Provide a readable XML-file structure and OpenMI XML schema definition (XSD) to store configuration related information
- [Req-DS2] – Provide support to save configuration related OpenMI objects in a readable way to disk according to the OpenMI XSD.
- [Req-DS3] - Provide support to read persistent XML-files, adhering to the OpenMI XSD, parse the information and turn them into instantiated OpenMI objects.
- [Req-DS4] – Provide object handling support to accommodate the display of objects and properties in front end applications
- [Req-DS5] – Provide other low level functionality to support an OpenMI implementation.

1.3 Scope of this document

This report contains the technical documentation of the org.OpenMI.DevelopmentSupport namespace. The technical documentation addresses the design as well implementation issues

1.4 Readership

This document is meant for code developers who have to implement, extend or maintain the source code of the OpenMI Software Development Kit. In order to understand this document, one needs to have basic understanding of model linking, object-orientation and UML-notation (particularly class diagrams and sequence diagrams). Within the text, the following style-convention is applied:

- OpenMI interface
- OpenMI method
- OpenMI property
- OpenMI argument

2 OpenMI DevelopmentSupport: Concepts

2.1 The OpenMI linking mechanism

OpenMI is a pull-based pipe and filter architecture, consisting of communicating components (providers and acceptors), which exchange data in a pre-defined way and in a pre-defined object format. Sometimes, this type of architecture is also referred to as a context based request-reply architecture, in which the context (i.e. the instantiated component) processes and replies to the requests in synchronized order.

Data exchange in OpenMI is based on direct model access in the same thread. No data is exchanged using persistent files. In addition, the entire specification of the links can be done at run-time. For convenience purposes however, it might be useful to support persistent storage of meta-data, enabling system configuration without actual access to the computational cores.

2.2 Task description of the OpenMI Development Support

Although the OpenMI architecture is fully memory based, the implementation accommodates the persistent storage of objects in XML. This functionality is specifically utilized so enable reuse of configurations of linked OpenMI components. To facilitate the development of such functionality in a generic way, a general XML parser has been developed. The org.OpenMI.DevelopmentSupport package offers general routines for reading and writing XML files and all related necessary functions. Note that this package doesn't depend on any OpenMI package.

The main task of the org.OpenMI.DevelopmentSupport namespace is to provide supporting classes for the Software Development Kit:

- enabling persistent storage and retrieval of configuration related OpenMI objects;
- other low level functionality, amongst others to convert the time information expressed in the Georgian Calendar to the Modified Julian Data.

To address the first bullet, a generic, customizable XML parser has been implemented using the introspection mechanisms as available in development language such Java (i.e. introspection) and .NET (System.Reflection).

2.3 Conceptual design of org.OpenMI.DevelopmentSupport

Figure 2 illustrates the main concept of the org.OpenMI.DevelopmentSupport functionality, namely transformation of object relations via a hierarchical data structure representation into an XML-tag representation (and vice versa). All public properties of any object can be saved to an XML file and later parsed into an object. Meta info is used heavily to customize the layout of the XML file.

Generally, the layout of the XML file is as follows. Each public property of an object is written as an XML element or an XML attribute. Attributes will only be used for primitive types (int, string, double etc.) and enumerations. The object representing a property value is written with all his public properties, in this way creating a tree in the XML file. For properties representing a collection (i.e. implementers of the IList and IDictionary interface, such as array lists and hash tables), all members are written as child XML elements.

Additionally, class type information is written under the XML attribute "type". This is only necessary if 1) the class of the object representing a property value differs from the property definition in the parent class (this occurs when a subclass is assigned to the property) or 2) in collections, MetaInfo doesn't define the desired member class type.

If the data structure isn't hierarchical, i.e. there are objects which represent property values or collection members more than once, only the first time in the xml file they are written completely. Next times only a reference to the object is written. MetaInfo must be used to set the reference definition.

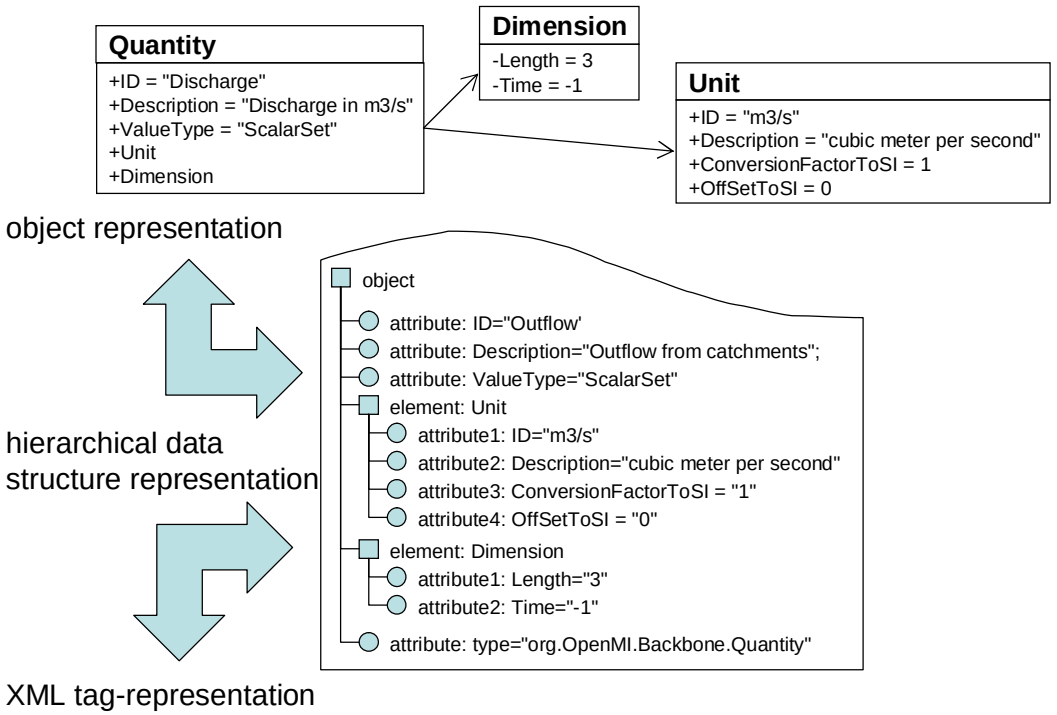

Figure 2 Transformation of object representation

This transformation process is the basic functionality required for parsing XML-files or serializing objects in memory. The major class for this transformation process in called XmlFile, which has two basic functions, namely Read(object, filename) and Write (filename).

The XmlFile-class is supported by a number of other classes and interfaces (see Section 3.2) of which MetaInfo is essential to customize the hierarchical data structure representation and the IAggregate interface is essential to contain the actual property values of the object in the hierarchical data structure.

2.4 The org.OpenMI.DevelopmentSupport namespace

2.4.1 Packages

The org.OpenMI.DevelopmentSupport namespace consists of one package.

2.4.2 Relations to other namespaces

The org.OpenMI.DevelopmentSupport namespace provides object handling and persistent IO functionality for the org.OpenMI.Utilities.Configuration namespace. The og.OpenMI.Utilities.configuration.Xml package contains the OpenMI specific MetaInfo for the

customization of the XML-parser. org.OpenMI.DevelopmentSupport provides calendar conversion functionality is used by the org.OpenMI.Utilities.Wrapper package.

The org.OpenMI.DevelopmentSupport namespace is independent of any other OpenMI namespace (see Figure 3).

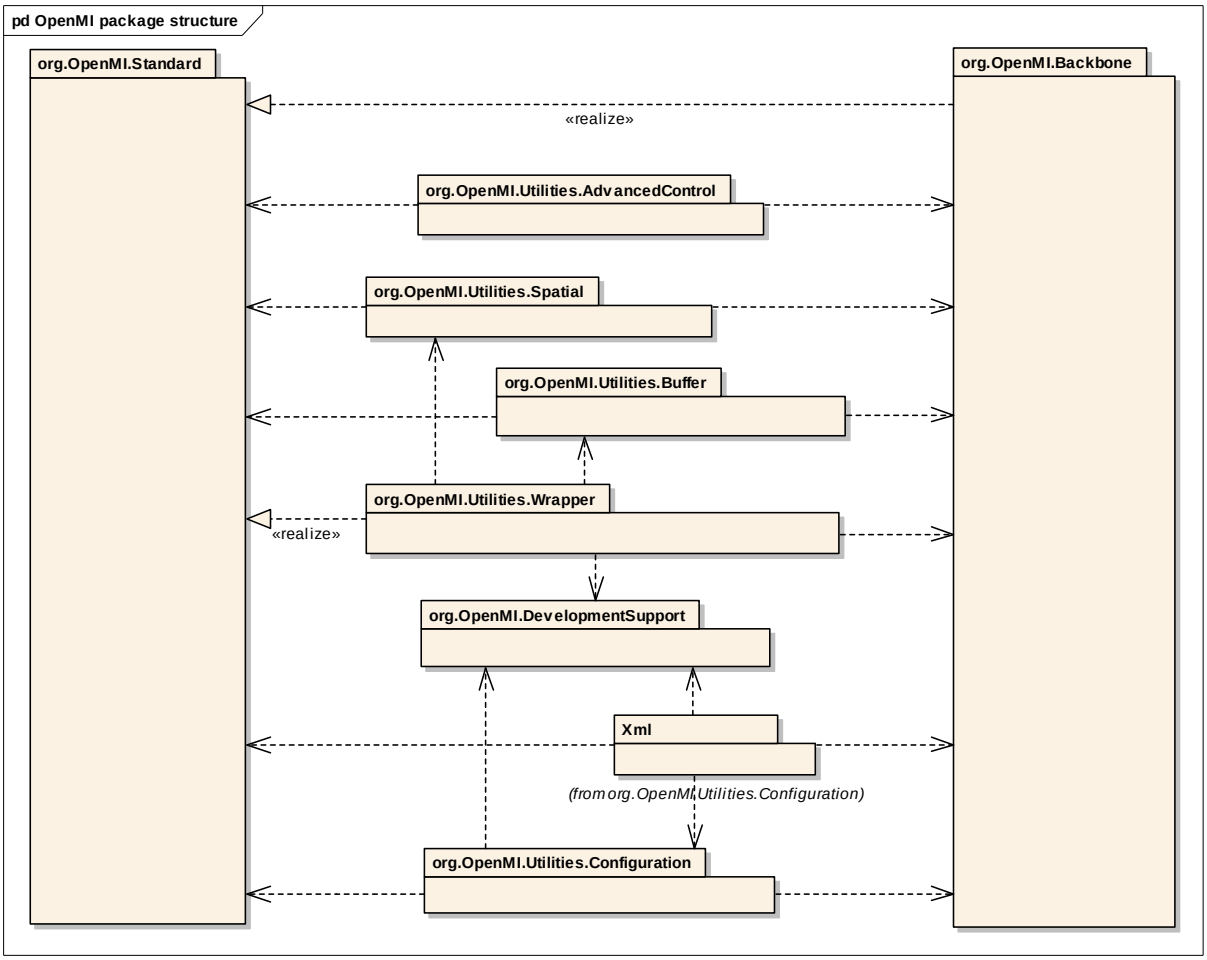

Figure 3 Overview of packages depending on org.OpenMI.DevelopmentSupport

3 The org.OpenMI.DevelopmentSupport package

3.1 Design considerations

As indicated in Section 2.3, the basic concept underlying the XML-parsing functionality is the transformation of object relations via a hierarchical data structure representation into an XML-tag representation (and vice versa). This transformation process is the basic functionality required for parsing XML-files or serializing objects in memory. The major class for this transformation process in called XmlFile, supported by a number of other classes and interfaces (see Section 3.2) of which MetaInfo is essential to customize the hierarchical data structure representation and the IAggregate interface is essential to contain the actual property values of the object in the hierarchical data structure.

3.1.1 Type information

When parsing xml files, XmlFile must have information about which type to instantiate. XmlFile tries to derive this information from the MetaInfo or from the type definition of the class. For example, when XmlFile encounters the xml element describing the Unit in the Quantity, it looks in the Quantity class definition to see what the type is of property Unit (the reflection package is used for this). Then XmlFile knows what type to instantiate for the Unit. In another case it may be possible that it can't derive the right type in this way, for example the ElementSet property in an ExchangeItem is defined as IElementSet. In that case an xml attribute "type" is used, which stores the right class to instantiate (this could be org.OpenMI.Backbone.ElementSet).

3.1.2 Using references to prevent duplication of similar objects

Sometimes similar objects are referred to by several objects, for example two LinkableComponent-objects referring to the same Link object, one being the provider the other being the acceptor. When parsing the xml file, the XmlFile should know that it should instantiate only one Link-object, although it is encountered more than once (each time it parses an xml element representing a LinkableComponent). This is solved in the following way: a difference is made between (i) an xml element, which corresponds with an object to be instantiated, and (ii) an xml element, which refers to an already instantiated object. In the first case the xml element is written out fully, in the second case only the identifier(s) of the object are written. MetaInfo is used to inform the XmlFile what properties identify an object.

Xml files may refer to other xml files where referred information is stored. For example, a composition xml file has references to model files, where models are stored. The file to be read is stored in the xml attribute "file", which holds a relative path to the referred file.

As said, properties of objects are represented in the xml file. Besides that, also elements in lists (strictly spoken, implementers of IList) are represented as an xml element and key value pairs in hash tables (strictly spoken, implementers of IDictionary).

3.1.3 Information captured in methods

XmlFile is very useful when all information in objects is accessible via properties, but this isn't always the case. Sometimes methods are used to store or retrieve information, having some arguments which are not known to XmlFile. In that case XmlFile may still be used.

The solution is as follows:

Instead of accessing an object directly, an intermediate object is accessed which must implement the interface IAggregate. This interface enables XmlFile to access all necessary information in the underlying object in a generic way. Each object is accessed via an aggregate, mostly via a default aggregate, which uses introspection to access the object properties. But it is possible to write another implementation of IAggregate, which contains dedicated code (for specific class) which exposes all information of an object (including the information captured in methods) as properties. This procedure has been adopted to reproduce all information from the ElementSet class and the Element class (both are part of the org.OpenMI.Backbone package). In MetaInfo the aggregate class should be registered for the object class (not necessary for the default aggregate).

3.2 Static View

3.2.1 XmlFile

The XmlFile class contains generic methods to read and write XML files (see Figure 4). All public properties of any object can be saved to an XML file and later parsed into an object. MetaInfo is used heavily to customize the layout of the XML file. The general approach to represent an object with its property values has been discussed in Section 2.3.

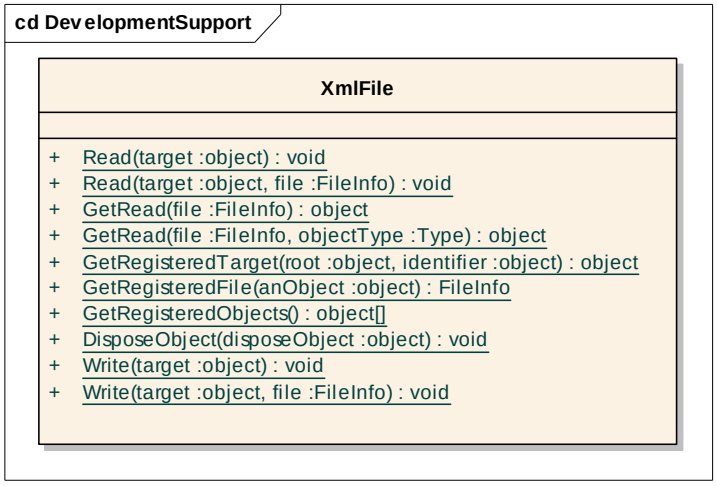

Figure 4 Methods of class XmlFile

XmlFile Methods

Method	Notes
Read (*object*)	Reads an object from file The file to read should have been registered with the object earlier by a read or write action
Read (*object, file*)	Reads an object from file
GetRead (*file*) *: object*	Reads and creates an object from a given file
GetRead (*file, type*) *: object*	Creates and reads an object from a given file. The class type of the object to be created is passed, which might not be specified in the file
GetRegisteredTarget (*root, key*) *: object*	Gets an object from the XmlFile internal registration given the key. Since object keys are unique within the scope of one file, the root of the file is passed.

GetRegisteredFile (*object*) *: file*	Gets the file associated with an object. The object should be the root of some file, i.e. associated with the top xml element.
GetRegisteredObjects () *: list*	Gets a list of all objects, which are known to be the root of a file. The root of a file is the object associated with the top xml element.
DisposeObject (*object*)	Removes an object from the XmlFile registration. To be used for releasing memory
Write (*object*)	Writes an object to an xml file The registered file of the object will be used as xml file. This is the file to which the object was written or read from in an earlier stage
Write (*object, file*)	Writes an object to an xml file

Typical usage of XmlFile

```
// Writes composition to file
Composition composition = new Composition ();
XmlFile.Write (composition, new FileInfo("composition.xml");

// Reads composition from file
Composition composition = (Composition) XmlFile.GetRead (new
FileInfo("composition.xml"));
```

3.2.2 IAggregate and related classes

An aggregate is a class that holds a hierarchical data representation of an object. Its generic nature accommodates easy methods to obtain information about the type and the properties of the object represented, including properties hidden in method calls. The concept of an Aggregate is based on common functionality that enables interrogation and serialization of an object. The IAggregate interface (see Figure 5) is defined as a platform independent interface to functionality that can be implemented in .NET using System.Reflection or implemented in Java using introspection.

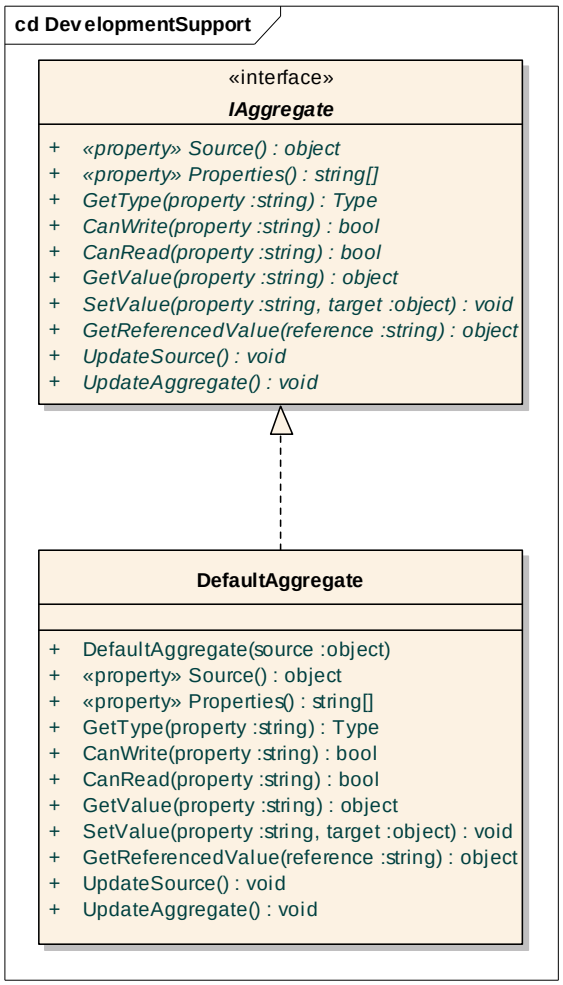

Figure 5 IAggregate interface and default implementation

IAggregate Properties and Methods

Property / Method	Notes
Source *: object*	The underlying object which holds the actual information.
Properties *: string array*	List of properties which can be queried in a generic way
GetType (*property*) *: type*	Gets the class type of one of the properties
CanWrite (*property*) *: bool*	Tells whether a value can be assigned to the property
CanRead (*property*) *: bool*	Tells whether a value can be retrieved from the property
GetValue (*property) : object*	Gets the value of a property
SetValue (*property, object*)	Sets the value of a property
GetReferencedValue (*property*) *: object*	Gets a property value by reference. A reference isn't necessarily a property, but can be any string, as long as it can be interpreted by the aggregate
UpdateSource ()	Tells the aggregate to process all information passed with SetValue calls
UpdateAggregate ()	Tells the aggregate to prepare for subsequent GetValue calls

DefaultAggregate Properties and Methods

Property / Method	Notes
DefaultAggregate (*object*)	Constructor which gets the underlying source object
Source *: object*	Gets the underlying source object
Properties *: string array*	Gets a list of all properties defined in the class type of the source. Reflection is used to get this list.
GetType (*property*) *: type*	Gets the class type of a property by reflection
CanWrite (*property*) *: bool*	Indicates whether a property can be written to. Reflection is used.
CanRead (*property*) *: bool*	Indicates whether a property can be read. Reflection is used.
GetValue (*property*) *: object*	Gets a value for a certain property. Reflection is used.
SetValue (*property, object*)	Sets a value for a certain property. Reflection is used
GetReferencedValue (*property*) *: object*	Gets a referenced value, i.e. a value corresponding with a reference string within the scope of the source. Implementation is delegated to XmlFile.GetRegisteredTarget.
UpdateSource ()	Intended for updating the source after various SetValue calls. Takes no action, because all SetValue calls are delegated directly to the source object with reflection.
UpdateAggregate ()	Intended for updating the aggregate before various GetValue calls. Takes no action, because all GetValue calls are delegated directly to the source object with reflection.

Typical usage of an aggregate by XmlFile. A new object is created and a value for a property is assigned to it.

```
// Create a new instance for a known type
object someObject = ObjectSupport.GetInstance(someType);

// Create an aggregate for the new object
IAggregate aggregate = new DefaultAggregate(someObject);

// If writable, set a value for a known property
if (aggregate.CanWrite(property))
{
      aggregate.SetValue (property, propertyValue);
}
```

3.2.3 Meta Info

Not all properties of a hierarchical data structure can be understood correctly without extra information. This extra information about an object is provided via the MetaInfo class (see Figure 6). The MetaInfo class registers extra properties about objects. These objects can be any kind of object, even class types. Usually there is some configuration part of an application, setting these properties, and some generic utilities, reading these properties. In this way the generic utilities will customize their behaviour to the demands of that application. For example, this class is heavily used by xml file to configure its reading and writing formats.

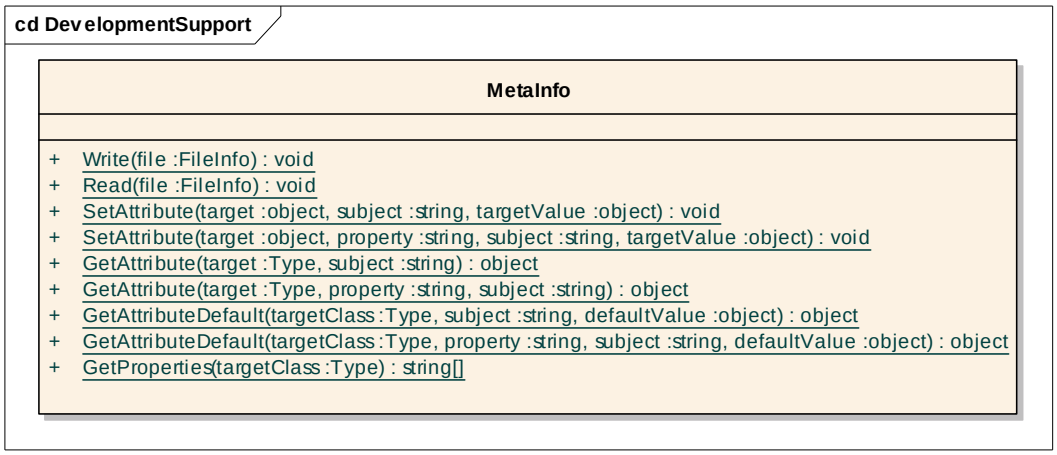

Figure 6 Methods of class MetaInfo

Method	Notes
Write (*file*)	Writes all metainfo to a file
Read (*file*)	Reads all metainfo from file
SetAttribute (*class type, subject, value*)	Stores a value for a class (usually as class type or string). Values for different subjects can be stored.
SetAttribute (*class type, property, subject, value*)	Stores a value for a class and property. Values for different subjects can be stored.
GetAttribute (*class type, subject*) *: object*	Gets the stored information for a class concerning a certain subject. Not only the class is examined, but also all superclasses and implemented interfaces.
GetAttribute (*class type, property, subject*) *: object*	Gets the stored information for a class and property concerning a certain subject. Not only the class is examined, but also all superclasses and implemented interfaces.
GetAttributeDefault (*class type, subject, default*) *: object*	Gets the stored information for a class concerning a certain subject. Not only the class is examined, but also all superclasses and implemented interfaces. If the value is not found, the default value is returned.
GetAttributeDefault (*class type, property, subject, default*) *: object*	Gets the stored information for a class and property concerning a certain subject. Not only the class is examined, but also all superclasses and implemented interfaces. If the value is not found, the default value is returned.
GetProperties (*class type*)	Gets a list of all properties in a class for which a value has been stored. All superclasses and implemented interfaces of the class are examined too.

The following gives a small example how to set and get information using MetaInfo

```
// Sets the value true to subject XmlKey of property ID in class Element
MetaInfo.SetAttribute (typeof(Element), "ID", "XmlKey", true);

// Gets the same value and supplies a default value if not found
```

```
if ((bool) MetaInfo.GetAttributeDefault (typeof(Element)), "ID", "XmlKey",
false)) DoSomething();
```

The following table explains the MetaInfo properties utilized by the XmlFile class.

Table 1 Explanation of MetaInfo properties of class XmlFile

Property	*Description*
ObjectAggregate	**queried for:** each class type encountered during reading and writing **expected type:** string **default value:** DefaultAggregate **meaning:** Specifies the class type of an aggregate. For each object identified by an xml element an aggregate is instantiated. The aggregate serves as an "in between" object between the object and XmlFile. NOTE: XmlFile expects that the aggregate has a constructor with one argument, the underlying object.
XmlFile	**queried for:** each class type encountered during writing **expected type:** Boolean **default value:** false **meaning:** Indicates whether an object of the specified type should be written as a separate XML file. An XML file and file name can be generated if not known to the XmlFile object. A relative path to this XML file is written in the parent XML file. On reading, if a reference is encountered to such an XML file, this referred file is read immediately.
XmlSchema	**queried for:** the class type encountered at the top of an xml file during reading **expected type:** string **default value:** null **meaning:** The name of the xsd file against which xml validation is performed. When this value is null, no xml validation is performed.
XmlNameSpace	**queried for:** the class type encountered at the top of an xml file during writing **expected type:** string **default value:** null **meaning:** The xml namespace. When validation is performed against a schema (xsd file), the xml file must have an xml namespace. For correct validation the xml namespace must be the same as the one specified in the xsd file.
XsdPackage	**queried for:** the class type encountered at the top of an xml file during reading **expected type:** string **default value:** null **meaning:** Indicates the assembly which contains the xsd file (denoted with XmlSchema).
XmlElement	**queried for:** each property in an object during writing **expected type:** Boolean **default value:** true **meaning:** Indicates whether a property should be written to XML.

Property	*Description*
XmlKey	**queried for:** each property in an object during writing **expected type:** Boolean **default value:** false **meaning:** Indicates whether a property should be written to XML if the corresponding object has been written to the XML file before. This occurs when the data structure isn't hierarchical. Only the first time it is encountered on writing, it is written out completely. Next times only a reference is written to the first occurrence. The key of the object is used as reference. The key of the entire object is the combination of all properties denoted as XML key. This key should be unique per class type per XML file.
XmlAllowGeneration	**queried for:** each property in an object during writing, which is denoted as an xml key. **expected type:** Boolean **default value:** true **meaning:** Indicates whether generation of a key value is allowed if the property value is empty (i.e. null or a string of length zero or a string only containing spaces). If generation is to be carried out, a unique number is assigned to the property value. The property type is expected to be an integer or a string.
XmlName	**queried for:** each property in an object during writing and reading **expected type:** string **default value:** the property name as defined in the class **meaning:** Provides the name for the XML element or XML attribute in the XML file for a property.
XmlRefName	**queried for:** each property in an object during writing and reading, if the object is or will be written as a reference to a prior object in the XML file. **expected type:** string **default value:** the XML name **meaning:** Provides the name for the XML element or XML attribute in the XML file for a property,.
XmlItemName	**queried for:** each member of a collection during writing **expected type:** string **default value:** the single name derived from the XML name of the collection object, e.g. the single name of the collection name "nodes", "node list", "node set" or "node collection" will be "node". **meaning:** Provides the name for the xml element in the xml file for a collection member.
XmlRefItemName	**queried for:** each member of a collection during writing, if the member will be written as a reference to a prior object in the XML file. **expected type:** string **default value:** the single name derived from the XML name of the collection. **meaning:** Provides the name for the XML element in the XML file for a collection member.
XmlItemType	**queried for:** each collection property in an object during reading and writing **expected type:** string

Property	*Description*
	default value: null **meaning:** Defines the class type of collection members. If set, class type information can be omitted during writing.
XmlTypeAlias	**queried for:** each type definition encountered in the XML file during reading **expected type:** string **default value:** the same type definition. **meaning:** Defines the class name (including namespaces) to be instantiated for a type definition in the XML file. This is useful if a class has been renamed and XML files still contain the old class name.
XmlParent	**queried for:** each property in an object during writing **expected type:** object **default value:** null **meaning:** Defines the property of an object, which functions as its parent, i.e. the xml parent element in the xml file. For example, the IElementSet has “Parent” for this value. When the aggregate of the element set is asked for the property “Parent”, it searches for the linkable component which owns the element set. If the value is null, XmlFile assumes that the object which has this object as a property is the parent.
XmlIndex	**queried for:** each property in an object during writing **expected type:** integer **default value:** 1000 **meaning:** Defines a sorting number of the XML element. During write, XML child elements are sorted according to this sorting number, if equal alphabetically.

3.2.4 Support classes

To accommodate proper handling of data, a number of support classes have been developed (see Figure 7). The ObjectSupport class provides generic functions in respect with object handling. The FileSupport class provides generic functionality to handle relative file paths.

Figure 7 Overview of Support classes

ObjectSupport Methods

Method	Notes
GetCopy (*source*) : *copied object*	Gets a deep copy of a specified object. Deep copying copies all primitive and enumeration properties and the properties for which MetaInfo "ObjectCopy" is set to true
GetCopy (*source, path*) *: copied object*	Gets a deep copy of a specified object and copies referenced files Deep copying copies all primitive and enumeration properties and the properties for which MetaInfo "ObjectCopy" is set to true If files are encountered, they are copied too. A path is supplied where copied files will be placed. The path is relative to the original location of the files
Copy (*source, copied object*)	Deep copies all properties of source into the properties of target Deep copying copies all primitive and enumeration properties and the properties for which MetaInfo "ObjectCopy" is set to true

Method	Notes
Copy (*source, copied object, path*)	Deep copies all properties of source into the properties of target, including files Deep copying copies all primitive and enumeration properties and the properties for which MetaInfo "ObjectCopy" is set to true. If files are encountered, they are copied too. A path is supplied where copied files will be placed. The path is relative to the original location of the files
CopyFile (*file, path*) *: copied file*	Gets a copy of a file. If the file doesn't exist, the copied file will not exist neither. A relative path is supplied where the copied file will be placed.
LoadAssembly (*assembly name*)	Loads an assembly. The assembly name can be either a full path to a file or a full or partial name of an assembly registered in the GAC. An empty assembly name is ignored.
IsLoadedAssembly (*assembly*)	Tells whether an assembly has been loaded already
GetType (*class name*) *: type*	Gets the class object given a string describing the class. The assemblies loaded with LoadAssembly are examined
GetInstance (*class name*) *: object*	Creates a new object. Types with an argumentless constructor can be created this way
GetInstance (*class name, base value*) *: object*	Creates a new object using a base value (e.g. a string with its value). This value is passed as argument to the constructor. Normally primitives, enumerations and some value types can be instantiated this way. Also types with constructors having one argument can be instantiated.
GetInstance (*type*) *: object*	Creates a new object. Types with an argumentless constructor can be created this way
GetInstance (*type, base value*) *: object*	Creates a new object using a base value (e.g. a string with its value). This value is passed as argument to the constructor. Normally primitives, enumerations and some value types can be instantiated this way. Also types with constructors having one argument can be instantiated.
GetInstance (*type, base value, culture*) *: object*	Creates a new object using a base value (e.g. a string with its value). This value is passed as argument to the constructor. Normally primitives, enumerations and some value types can be instantiated this way. Also types with constructors having one argument can be instantiated. Culture info used for parsing the base value.

FileSupport Methods

Method	Notes
GetRelativePath (*directory, file*) *: string*	Gets the relative path from a starting directory to a file
GetRelativePath (*directory, directory*) *: string*	Gets the relative path from a starting directory to a directory
ExpandRelativePath (*directory, relative path*) *: file*	Expands a relative path to a file
ExpandRelativeDirectory (*directory, relative path*) *: directory*	Expands a relative path to a directory

CollectionSupport Methods

Method	Notes
ContainsObject (*collection, object*) *: bool*	Indicates whether an object is contained by a collection by comparing references. The Equals method is not used, in contrary with the Contains method in the ArrayList.
ToArray (*collection*) *: array object*	Converts a collection to an array

3.2.5 Calendar Converter class

Within OpenMI, time is represented as a Modified Julian Date, i.e. the number of days since November 17, 1858. As many models and user interfaces communicate in the Gregorian calendar (e.g. January 1, 2005), a class has been developed to convert calendar information, i.e. convert a gregorian DateTime object into a Modified Julian Data double and vice versa. This class currently is named CalendarConverter (see Figure 8).

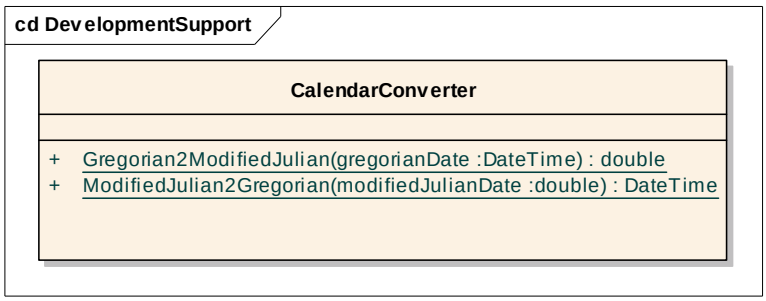

Figure 8 Methods of the class CalendarConverter

Method	Notes
Gregorian2ModifiedJulian (*DateTime*) *: double*	Converts a DateTime object to modified julian date
ModifiedJulian2Gregorian (*double*) *: DateTime*	Converts a modified julian date to a DateTime object

3.3 Dynamic view

3.3.1 Behaviour logic of XmlFile

As indicated, ‘read’ and “write’ functions of XmlFile are the main methods that are invoked by external software packages. The processing of information is done xml-element by xml-element. The logical behaviour of XmlFile can best be illustrated by activity diagrams showing the decision logic which determines the internal calls. The activity diagram of Figure 9 shows the algorithm how XmlFile reads an xml file.

XmlFile processes an xml element and then progresses to all xml sub elements. Each xml sub-element is processed in the same way (this is performed by a recursive call to read xml element). When an object is referenced in another unread xml file, first that file is read by a recursive call. When XmlFile comes to the conclusion that an xml element denotes a reference to an already existing object, a data store is accessed to retrieve this object. Identified by file, object identifiers and type of the object, the right instance can be retrieved. If it is not there, an exception is raised. When reading a fully written out xml element, the instantiated object is stored in the data store, because it may be referred to further down in the xml file. The data store is persistent during the lifetime of XmlFile.

The activity diagram of Figure 10 illustrates the algorithm for writing an xml-file. When writing an xml file, more or less the same structure is used as during reading an xml file. Now XmlFile processes all properties of an object and writes corresponding xml elements and attributes. A recursive call is used to write sub properties into xml sub-elements.

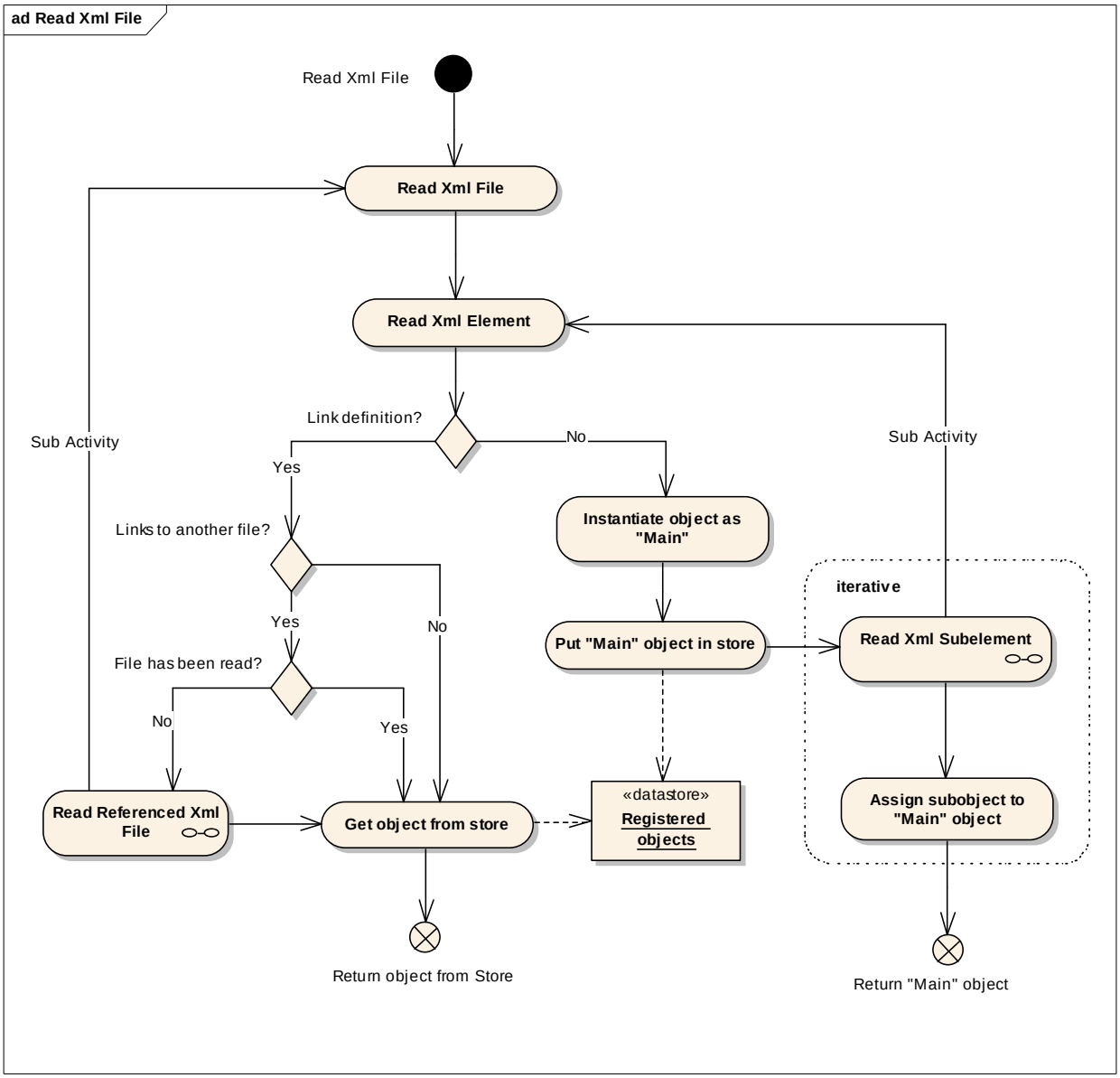

Figure 9 Reading an xml-file; behaviour logic of the class XmlFile

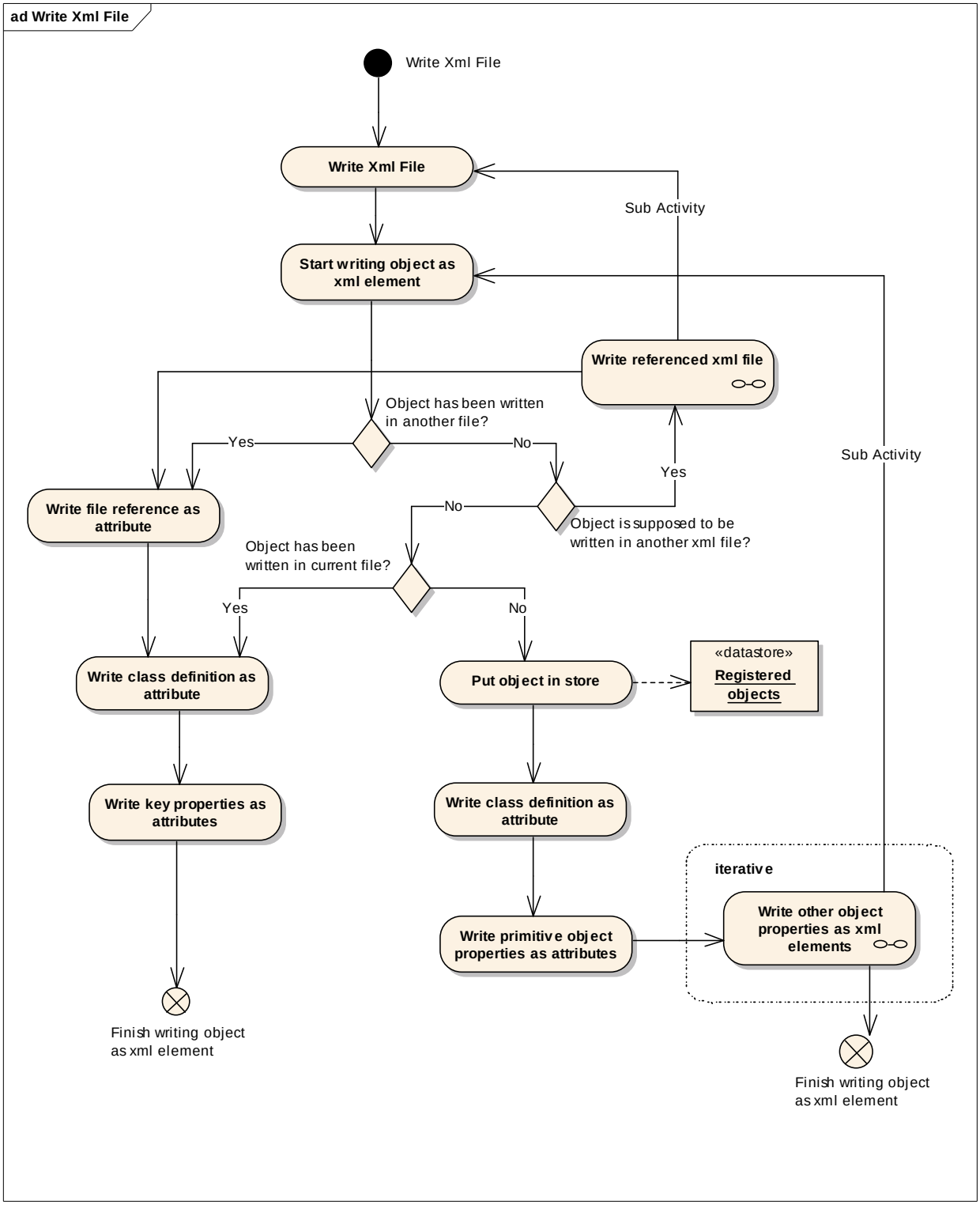

Figure 10 Writing an xml-file; logical behaviour of the XmlFile-class

3.3.2 Exceptions

Utilizing a generic parser for reading XML-files requires proper exception handling. Table 2 provides an overview of the exceptions generated by classes in the org.OpenMI.DevelopmentSupport package.

Table 2 Overview of exceptions generated by org.OpenMI.DevelopmentSupport

Class	Method	Exceptions
XmlFile	+Read(object) +Read (object, objectType) +GetRead (file) +GetRead (file, objectType)	Cannot find file to read Cannot resolve class type when an object for an xml element must be instantiated Cannot find class type Xml element holds a reference, but referenced object cannot be found Schema cannot be found although it has been specified in MetaInfo Validation error when xml file doesn't meet specified schema
XmlFile	+Write (object) +Write (object, file)	Cannot derive key for object which must be written as a reference. Probably MetaInfo is missing for the class and subject XmlKey
ObjectSupport	+GetType(ClassType)	Class cannot be found
ObjectSupport	+LoadAssembly (assemblyName)	Assembly cannot be found in the GAC
DefaultAggregate	+GetValue(property)	Internal exception raised by the source object when getting the value
DefaultAggregate	+SetValue(property, target object)	Internal exception raised by the source object when getting the value

3.4 Implementation remarks

3.4.1 C#-implementation

The correct implementation of all methods has been tested using dedicated unit tests in combination with the NUnit framework for testing.

3.4.2 Java-implementation

No Java-implementation is available.