Service Data Objects For C++Specification

Version 2.01, November 2005

Authors

John Beatty	BEA Systems Inc.
Stephen Brodsky	IBM Corporation
Raymond Ellersick	IBM Corporation
Michael Ho	Sybase, Inc.
Todd Little	BEA Systems, Inc.
Martin Nally	IBM Corporation
Pete Robbins	IBM Corporation.
Ed Slattery	IBM Corporation.
Colin Thorne	IBM Corporation.

Copyright Notice

License

The Service Data Objects Specification is being provided by the copyright holders under the following license. By using and/or copying this work, you agree that you have read, understood and will comply with the following terms and conditions:

Permission to copy and display the Service Data Objects Specification and/or portions thereof, without modification, in any medium without fee or royalty is hereby granted, provided that you include the following on ALL copies of the Service Data Objects Specification, or portions thereof, that you make:

1. A link or URL to the Service Data Objects Specification at these locations:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sdo/
- http:/www.sybase.com/sca

2. The full text of this copyright notice as shown in the Service Data Objects Specification.

IBM, BEA, Sybase (collectively, the “Authors”) agree to grant you a royalty-free license, under reasonable, non-discriminatory terms and conditions to patents that they deem necessary to implement the Service Data Objects Specification.

THE Service Data Objects SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THIS SPECIFICATION AND THE IMPLEMENTATION OF ITS CONTENTS, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT OR TITLE. THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE SERVICE DATA OBJECTS SPECIFICATION.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Service Data Objects Specification or its contents without specific, written prior permission. Title to copyright in the Service Data Objects Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

Status of this Document

This specification may change before final release and you are cautioned against relying on the content of this specification. The authors are currently soliciting your contributions and suggestions. Licenses are available for the purposes of feedback and (optionally) for implementation.

Table Of Contents

Introduction

Service Data Objects (SDO) is a data programming **architecture** and an **API**.

The main purpose of SDO is to simplify data programming, so that developers can focus on business logic instead of the underlying technology.

SDO simplifies data programming by:

- unifying data programming across data source types
- providing support for common application patterns
- enabling applications, tools and frameworks to more easily query, view, bind, update, and introspect data.

For a high-level overview of SDO, see the white paper titled "Next-Generation Data Programming: Service Data Objects" [3].

Key Concepts

The key concepts in the SDO architecture are the **Data Object**, the **data graph** and the **Data Access Services (DAS)**.

A Data Object holds a set of named properties, each of which contains either a simple data-type value or a reference to another Data Object. The Data Object API provides a dynamic data API for manipulating these properties.

The data graph provides an envelope for Data Objects, and is the normal unit of transport between components. Data graphs can track changes made to the graph of Data Objects. Changes include inserting Data Objects, deleting Data Objects and modifying Data Object property values.

Usually, data graphs are constructed from one of the following:

- Data sources such as XML files, Enterprise Java™ Beans (EJBs), XML databases and relational databases.
- Services such as Web services, Java Connector Architecture (JCA) Resource Adapters and Java Message Service (JMS) messages.

Components that can populate data graphs from data sources and commit changes to data graphs back to the data source are called Data Access Services (DAS). The DAS architecture and APIs are outside the scope of this specification.

Requirements

The scope of the SDO specification includes the following requirements:

1. **Dynamic Data API.** Data Objects often have typed interfaces. However, sometimes it is either impossible or undesirable to create interfaces to represent the Data Objects. One common reason for this is when the data being transferred is defined by the output of a query. Examples would be:
   - A relational query against a relational persistence store.
   - An EJBQL queries against an EJB entity bean domain model.
   - Web services.
   - XML queries against an XML source.
   - When deployment of generated code is not practical.

   In these situations, it is necessary to use a dynamic store and associated API. SDO has the ability to represent Data Objects through a standard dynamic data API.

2. **Support for Static Data API.** In cases where metadata is known at development time (for example, the XML Schema definition or the SQL relational schema is known), SDO supports code-generating interfaces for Data Objects. When static data APIs are used, the dynamic data APIs are still available. SDO enables static data API code generation from a variety of metamodels, including:
   - Popular XML schema languages.
   - Relational database schemas with queries known at the time of code generation.
   - Web services, when the message is specified by an XML schema.
   - JCA connectors.
   - JMS message formats.
   - UML models

   While code-generation rules for static data APIs is outside the scope of this core SDO specification, it is the intent that SDO supports code-generated approaches for Data Objects.

3. **Complex Data Objects.** It is common to have to deal with "complex" or "compound" Data Objects. This is the case where the Data Object is the root of a tree, or even a graph of objects. An example of a tree would be a Data Object for an Order that has references to other Data Objects for the Line Items. If each of the Line Items had a reference to a Data Object for Product Descriptions, the set of objects would form a graph. When dealing with compound data objects, the change history is significantly harder to implement because inserts, deletes, adds, removes and re-orderings have to be tracked, as well as simple changes. Service Data Objects support arbitrary graphs of Data Objects with full change summaries.

4. **Change Summary.** It is a common pattern for a client to receive a Data Object from another program component, make updates to the Data Object, and then pass the modified Data Object back to the other program component. To support this scenario, it is often

important for the program component receiving the modified Data Object to know what modifications were made. In simple cases, knowing whether or not the Data Object was modified can be enough. For other cases, it can be necessary (or at least desirable) to know which properties were modified. Some standard optimistic collision detection algorithms require knowledge not only of which columns changed, but what the previous values were. Service Data Objects support full change summary.

5. **Navigation through graphs of data.** SDO provides navigation capabilities on the dynamic data API. All Data Objects are reachable by breadth-first or depth-first traversals, or by using a subset of XPath 1.0 expressions.

6. **Metadata.** Many applications are coded with built-in knowledge of the shape of the data being returned. These applications know which methods to call or fields to access on the Data Objects they use. However, in order to enable development of generic or framework code that works with Data Objects, it is important to be able to introspect on Data Object metadata, which exposes the data model for the Data Objects. SDO provides APIs for metadata. SDO metadata may be derived from:
   - XML Schema
   - EMOF (Essential Meta Object Facility)
   - Java
   - Relational databases
   - Other structured representations.

7. **Validation and Constraints.**
   - Supports validation of the standard set of constraints captured in the metadata. The metadata captures common constraints expressible in XML Schema and relational models (for example, occurrence constraints).
   - Provides an extensibility mechanism for adding custom constraints and validation.

8. **Relationship integrity.**
   - An important special case of constraints is the ability to define relationships between objects and to enforce the integrity of those constraints, including cardinality, ownership semantics and inverses. For example, consider the case where an employee has a relationship to its department and a department inversely has a list of its employees. If an employee's department identifier is changed then the employee should be removed, automatically, from the original department's list. Also, the employee should be added to the list of employees for the new department. Data Object relationships use regular Java objects as opposed to primary and foreign keys with external relationships.
   - Support for containment tree integrity is also important.

NOTE the following areas are out of scope:

9. **Complete metamodel and metadata API.** SDO includes a minimal metadata access API for use by Data Object client programmers. The intention is to provide a very simple client view of the model. For more complete metadata access, SDO may be used in conjunction with common metamodels and schema languages, such as XML Schema [1] and the EMOF compliance point from the MOF2 specification [2]. Java annotations in JSR 175 may be a future source of metadata.

10. **Data Access Service (DAS) specification.** Service Data Objects can be used in conjunction with "data accessors". Data accessors can populate data graphs with Data Objects from back-end data sources, and then apply changes to a data graph back to a data source. A data access service framework is out of scope but will be included in a future Data Access Service specification .

Organization of this Document

This specification is organized as follows:

- **Architecture**: Describes the overall SDO system.
- **API**: Defines and describes the Programming API for SDO.
- **Generating Code from XML Schemas**: Shows how code is generated from XML Schemas (XSD).
- **Interface Specification**: Defines how interfaces are generated and used.
- **Serialization of DataObjects**: Defines how to serialize DataObjects.
- **Standard SDO Types**: Defines and describes the Standard SDO Types.
- **XML Schema to SDO Mapping**: Defines and describes how XML Schema declarations (XSD) are mapped to SDO Types and Properties.
- **Generation of XSD from SDO Type and Property**: Describes how to generate XSDs from SDO Types and Properties.
- **XPath Expression for DataObjects**: Defines an augmented subset of XPath that can be used with SDO for traversing through Data Objects.
- **Examples**: Provides a set of examples showing how SDO is used.
- **DataType Conversion Tables**: Shows the set of defined datatype conversions.

Architecture

The core of the SDO framework is the DataObject, which is a generic representation of a business object and is not tied to any specific persistent storage mechanism.

A data graph is used to collect a graph of related DataObjects. In SDO version 1.0 a data graph was always wrapped by a DataGraph envelope object, whereas in SDO version 2.0 a graph of DataObjects can exist outside of a DataGraph. When *data graph* is used as two lower case words, it refers to any set of DataObjects. When *DataGraph* is used as a single upper case word, it refers specifically to the DataGraph envelope object.

All data graphs have a single root DataObject that directly or indirectly contains all the other DataObjects in the graph. When all DataObjects in the data graph refer only to DataObjects in the data graph, then the data graph is called *closed*. *Closure* is the normal state for data graphs.

A data graph consists of:

- A single root DataObject.
- All the DataObjects that can be reached by recursively traversing the containment Properties of the root DataObject.

A closed data graph forms a tree of DataObjects, where the non-containment references point to DataObjects within the tree.

A data graph keeps track of the schema that describes the DataObjects. A data graph can also maintain a ChangeSummary, which represents the changes made to the DataObjects in the graph.

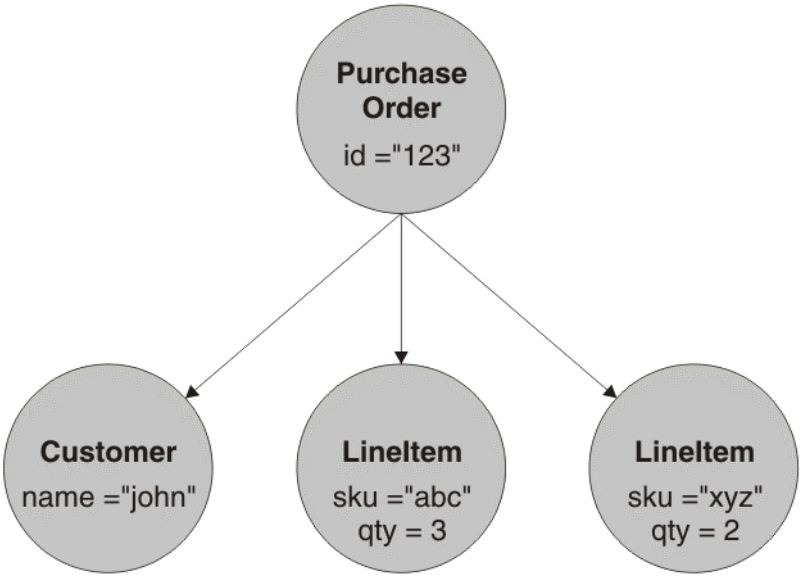


Figure 1: Data graph containing Data Objects

The standard way for an end user to get access to a data graph is through a Data Access Service (DAS). A DAS is a facility that provides methods to load a data graph from a store and to save a data graph back into that store. For example, an XML File DAS would load and save a data graph as an XML file and a JDBC DAS would load and save a data graph using a relational database. Specifications for particular DAS are outside the scope of this specification.

DAS typically uses a disconnected data architecture, whereby the client remains disconnected from the DAS except when reading a data graph or writing back a data graph. Thus, a typical scenario for using a data graph involves the following steps:

1. The end user sends a request to a DAS to load a data graph.
2. The DAS starts a transaction against the persistent store to retrieve data, creates a data graph that represents the data, and ends the transaction.
3. The DAS returns the data graph to an end user application.
4. The end user application processes the data graph.
5. The end user application calls the DAS with the modified data graph.
6. The DAS starts a new transaction to update the data in the persistent store based on the changes that were made by the end user.

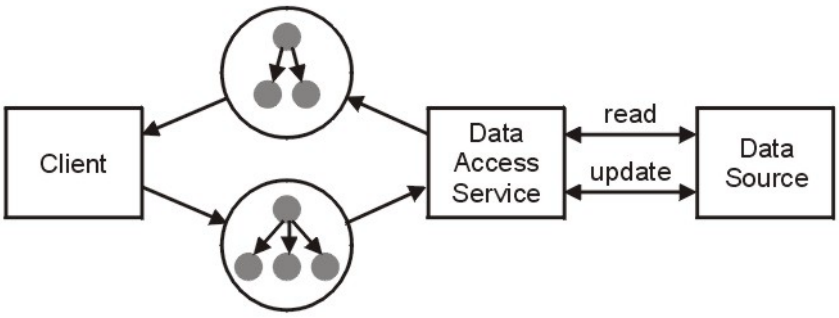


Figure 2: SDO's disconnected data architecture

Note that there are two distinct roles that can be identified among DataObject users: the client and the DAS writer.

The client needs to be able to traverse a data graph to access each DataObject and to get and set the fields in each DataObject. The client may also need to serialize and deserialize a data graph. Data graphs can be serialized to XML, typically by an XML DAS.

The DAS writer must be able to define a model for a data graph, create a new data graph, generate change history information, and access change history information. This specification’s focus is the perspective of the client.

A data graph contains a ChangeSummary that can be used to access the change history for any DataObject in the graph. Typically the ChangeSummary is empty when a data graph is returned from a DAS. If the client of the DAS makes modifications that change the state of the DataObjects, including creation and deletion, then a summary of changes can be recorded in the ChangeSummary.

If a client sends a modified data graph to a DAS, (the original DAS or a different one), then the DAS will check the data graph for errors. These errors include lack of closure of the data graph, values outside the lower and upper bounds of a property, choices spanning several properties or DataObjects, deferred constraints, or any restrictions specific to the DAS (for example, XML Schema specific validations). Closure means that any DataObject references, made within the graph of DataObjects, point to a DataObject that is in the graph. Usually, the DAS will report update problems by throwing exceptions.

It is possible that a data graph does not have closure, temporarily, while the contained DataObjects are being modified by an end user, through the DataObject interface. However, after all user operations are completed the data graph should be restored to closure. A DAS should operate only on data graphs with closure.

Memory Management

The C++ SDO implementation handles the life cycle of objects which it creates by using smart pointers and references.

In cases where the client application needs a read-only object, the API calls will return a reference to a constant object, thus there is no requirement on the part of the client to free any of these. Types and Properties are always read-only via the API, so these are visible only as const references.

Information concerning the allowable types and properties is held by a DataFactory , and all DataObjects created by the DataFactory require access to these types and properties. Therefore the life of a DataFactory encloses the life of all its created DataObjects.

The C++ API handles these life cycles by linking the DataObjects to the DataFactory with a reference counting smart pointer. This style of pointer is also handed out to the client application via the API. Pointers to DataObjects passed out to the client hold a reference count on the DataObject, and that DataObject will not be freed until all pointers to it go out of scope, or are intentionally deleted.

Similarly, each DataObject holds reference counting pointers to the DataFactory, and all its child DataObjects. No DataObject will be allowed to be freed until its reference count is zero.

See below for details of reference counting pointers.

STL

The current version of this specification exposes an API which is independent of the standard template library, and so uses “const char*” instead of std::string, and specially created collection classes instead of std::list etc.

A further API may be released later augmenting the calls which return “const char*” to return std::string, and replacing all of the lists DataObjectList, PropertyList etc with the equivalent either std::list or std::vector as appropriate.

Reference counting pointers.

These are a derived version of the familiar smart-pointer. The pointer class holds a real (dumb) pointer to the real data object. If reference counting pointer is copied, then a duplicate pointer is returned with the same dumb pointer to the same data object held within it. A reference count within the data object is incremented for each copy of the pointer, so only when all the pointers go out of scope will the data object be freed.

These reference counting pointers can usually be used exactly as you would use a normal pointer. The -> and * operators are overloaded such that they really act on the underlying data object.

There is, however, a limitation in terms of the syntax for checking for a null reference counting pointer. The pointer itself is never null, so checking for (p == 0) will fail. To provide a familiar syntax for null checking, the pointers have an override of the "!" operator, such that "if (!p)" will compile.

Pointers in SDO have the same name as the object at which they are pointing, with a suffix of Ptr. (E.g DataObjectPtr, SequencePtr…).

Example:

```
//  some code
{
  // create a data factory within some scope
  DASDataFactoryPtr  mdg  = DASDataFactory::getDataFactory();

  // use the ref counting pointer just as a normal pointer
  mdg->addType("myspace","Company");
  mdg->addType("myspace","Department");

  // Types are returned as references, so no need to free them.
  const Type& tc = mdg->getType("myspace","Company");

  // .. other setup code …
  //  get back a ref counting pointer to the data object…
  DataObjectPtr dor = mdg->create(tc);

  // dor now contains a reference to mdg, so even a delete mdg will
  // not get rid of mdg until dor is deleted, or goes out of scope…
     }
  // Both dor and mdg are out of scope, so no reference is held to either, and they will both be
```

freed.

C++ API

The SDO API is made up of the following interfaces that relate to instance data:

- DataObject – A business data object.
- DataGraph – An envelope for a graph of DataObjects.
- ChangeSummary – Summary of changes to the DataObjects in a DataGraph. - This API is of use to Data Access Service providers, and is not part of the client API.
- Sequence - A sequence of settings.

SDO also contains a minimal metadata API that can be used for introspecting the model of DataObjects:

- Type – The Type of a DataObject or Property.
- Property - A Property of a DataObject.
- PropertyList, TypeList and DataObjectList - Lists of Types and Properties

Finally, SDO has a number of helper interfaces and classes:

- SDORuntime - A utility for version information
- DataFactory - For creation of data objects
- CopyHelper - Shallow and deep copy
- EqualityHelper - Shallow and deep equality
- XMLHelper - Serialization to XML
- XMLDocument - Serialization to XML
- XSDHelper - Loading XSDs

The APIs are shown in figure 3 below.

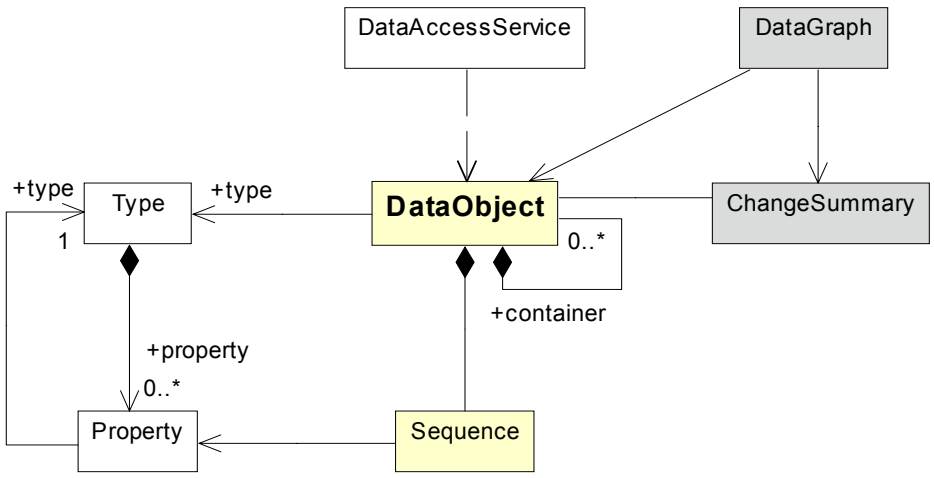


Figure 3: DataGraph APIs

DataObject

DataObjects represent business data. They hold their data in properties.

The DataObject interface is designed to make programming easier because it provides access to business data of all the common types and access patterns, such as name, index, and path.

The DataObject interface includes methods that:

- Get and set the properties of a DataObject.
- Query whether a Property is set.
- Create a new instance of a contained DataObject.
- Delete a DataObject from its container.
- Detach a DataObject from its container.
- Get the container of a DataObject and the containing property.
- Get the root DataObject.
- Get the DataGraph to which a DataObject belongs.
- Get the DataObject's Type.
- Get the DataObject's Sequence (if present).
- Get the DataObject's additional Properties (if present).

For many applications that do not use generated code, the DataObject interface is the only part of SDO that is used to write applications. For many applications that use generated code, the generated interfaces themselves are what is used. The other parts of SDO are primarily use-as-you-go.

DataObject Concepts

The C++ DataObject accessor methods are separated into getters and setters for each basic type , so there is a getBoolean, getString, getInteger etc, rather than just a get() method. The notation getXXX() is used to indicate any one of these accessor methods.

DataObjects can be thought of as falling into the following categories. The *open* and *sequenced* concepts can be used independently or together.

1. **Basic**. A DataObject has a field for each Property. The set of allowed Properties is defined by getType().getProperties(). Values are accessed through getXXX(property). Order within Properties is maintained but not across Properties.
2. **Open.** A DataObject is similar to the above plus it has tolerance for additional Properties. In XML this is equivalent to open (wildcard) content. The extra Properties are not part of getType().getProperties(). The Properties actually set in a specific DataObject are available through getInstanceProperties(). Values are accessed through getXXX(property). Order within Properties is maintained but not across Properties.
3. **Sequenced.** A DataObject is similar to a basic DataObject plus it has order within and across Properties. In XML this is equivalent to a DOM. When using XML, a Sequence

(see "Sequence" on page 41) represents the order of all the XML elements in the DataObject. Values are available through get(property) but order across Properties is maintained through the Sequence interface. getSequence() returns a Sequence of the XML elements for the case of XML. XML Attributes do not have the concept of order and are accessed through getXXX(property).

DataObject Values and Properties

DataObjects have data values assigned to Properties. For example, a purchase order DataObject could have the value (SDODate)3000 assigned to the orderDate property. Values for the orderDate property can be returned or changed using the getDate("orderDate") and setDate("orderDate") accessors on the DataObject. When code is generated, values can also be accessed through getOrderDate() and setOrderDate() methods on a PurchaseOrder interface.

On the DataObject interface, values can be accessed using the name of the property with getXXX(String path), with the index of the property, or directly with a Property object. The getXXX(String path) methods on DataObject work with the alias names as well as the property names names in the path. The path can be just the name of the property, or it can be a path expression based on a subset of XPath.

Type Conversion

Sometimes the Type of a Property is different than the most convenient type for use in an application program. For example, when displaying a integer quantity in a user interface, the string representation is more useful than the int. The method getCString("quantity") for accessing an int quantity property conveniently returns the value as a null terminated C String. This simplifies a common task in many applications.

When a DataObject's typed accessors getXXX() and setXXX() are invoked, a type conversion is necessary if the value is not already an instance of the requested type XXX. Type conversion is automatically done by a DataObject implementation. An implementation of SDO is expected to convert between any data type and the set defined in DataObject, with possible loss of information. The supported data type set is defined in the SDO DataTypes section. These types include:

- C++ primitives
- Date and time types
- URI

The supported conversions are specified in "DataType Conversions" on page 145.

Many-valued DataObject Properties

A Property can have one or many values. If a Property is many-valued then property.many is true and getXXX(property) always returns a DataObjectList.

DataObject methods with a return type of DataObjectList, on the DataObject interface or generated, return empty lists rather than null when there is no value. Returned Lists actively represent any changes to the DataObject's values.

The getList(property) accessor is especially convenient for many-valued properties. If property.many is true then setList(property, value) requires that "value" be a DataObjectList . For many-valued Properties, getList() returns a List containing the current values. Updates through the List interface operate on the current values of the DataObject immediately. Each access to get() or getList() returns the same List object.

Determining whether a Property is Set

For many-valued properties, isSet(property) returns:

- True, if the List is not empty.
- False, if the List is empty.

For single-valued properties, isSet(property) returns:

- False, if the Property has not been set(), or has been unset().
- True, if the current value is not the Property's default.
- For the remaining cases the implementation may decide between either of the following policies:
  - Any call to set() without a call to unset() will cause isSet() to return true. After set(property, property.getDefault()), isSet(property) returns true.
  - The current value is compared to the default value and isSet() returns true when they differ. After set(property, property.getDefault()), isSet(property) returns false.

The unset(property) accessors can be thought of as clearing out a single property, so that isSet(property) returns false and get(property) returns the default. The delete() method unsets all the DataObject's properties except for those marked read-only. After unset(), get(property) returns the default; which in the case of a many-valued Property is an empty List.

Note that attempts to modify read-only properties (using set, unset or delete) cause an exception.

Containment

DataObjects in a data graph are arranged in a *tree* structure. One DataObject forms the *root* of the tree and the other DataObjects make up the nodes of the tree.

The tree structure is created using *containment references* which start at the root DataObject. The root DataObject refers to other DataObjects, which can refer to further DataObjects. Each DataObject in the data graph, except for the root DataObject, must have a containment reference from another node in the tree. Each DataObject in the graph keeps track of the location of its containment reference.

It is possible for a data graph to have non-containment references. These are references to DataObjects which are part of the same data graph, (the referenced DataObjects must be part of the same tree), but these references do not affect the tree structure of the data graph.

Both containment and non-containment references are Properties of a DataObject. The Type of the Properties is any DataObject Type.

Whether a particular DataObject reference Property is a containment reference or a non-containment reference is defined by the data model for the data graph, for example the XSD which defines the data types for an XML document. This cannot be changed once the data model has been defined. You can query whether a particular reference is a containment reference accessing property.containment.

A container DataObject is one that contains other DataObjects. A DataObject can have a maximum of one container DataObject. If a DataObject has no container, it is considered to be a root DataObject.

Simple navigation, up and down the DataObject containment tree, is provided by getContainer() and getContainmentProperty(). The getContainer() method returns the parent DataObject and the getContainmentProperty() method returns the Property of the container that contains this object. A DataObject can be removed from its container, without making any other changes, using the detach() method.

Containment is managed. When a DataObject is set or added to a containment Property, it is removed from any previous containment Property. Containment cannot have cycles. If a set or add would produce a containment cycle, an exception is thrown.

Creating and Deleting DataObjects

The create methods create a DataObject of the Type of the Property, or the Type specified in the arguments, and add the created object to the Property specified. If the DataObject's Type is a sequenced type (that is, if getType().isSequenced() is true) then the created DataObject is put at the end of the Sequence. If the Property is single-valued, the Property is set to the created object. If the Property is multi-valued, the created object is added as the last object. Only containment properties may be specified for creation. A created object begins with all its properties unset.

The delete() method unsets all the DataObject's non-readonly properties. The delete() method will also remove the DataObject from its containing DataObject if the containment Property is not read-only. All DataObjects recursively contained by containment properties will also be deleted.

If other DataObjects have one-way, non-containment properties that refer to deleted DataObjects, then these references are not modified. However, these properties can need changing to other values, in order to restore closure to the data graph. A deleted DataObject can be used again, have its values set, and be added into the data graph again.

Sequenced DataObjects

A DataObject can be of a sequenced or unsequenced type (see Sequence). The getType().isSequenced() method tells you whether the DataObject's Type is sequenced or not.

If a DataObject's Type is sequenced then getSequence() returns that Sequence, otherwise getSequence() returns null.

The Sequence of a DataObject corresponds to the XML elements representing the values of its properties. Updates through DataObject, and the Lists or Sequences returned from DataObject, operate on the same data.

Returned Sequences actively represent any changes to the DataObject's values.

Open Content DataObject Properties

DataObjects can have two kinds of Properties:

1. Those specified by their Type (see Type)
2. Those not specified by their Type. These additional properties are called *open content*.

Properties which are specific to a DataObject's Type are returned in a PropertyList by getType().getProperties().

Open Content DataObject Properties

DataObjects can have Properties beyond those specified by their Type when either:

1. Handling XML open or mixed content.
2. Encountering new Properties dynamically.

Open content Properties are allowed only when Type.open is true. Some Types set open to false so they do not have to accept additional Properties.

A Property is from open content if it appears in getInstanceProperties() but not in getType().getProperties(). If a Property is from open content then isSet(property) must be true.

Properties with DataType Types may return different objects as long as equals() is true. For mutable data values (Date and List of Strings for example), modification of those values directly is implementation dependent.

All Properties currently used in a DataObject are returned, in a read-only PropertyList, when you invoke getInstanceProperties(). This includes properties that are open content. The order of the Properties begins with all the getType().getProperties() whether set or not; the order of the remaining Properties is determined by the implementation.
The property name can be used to find the corresponding Property active on the DataObject within the instance properties by calling getProperty().

In order to set an open content value when that Property is not set (it does not appear in getInstanceProperties()), a set or create accessor on DataObject, or add on List or Sequence, with a Property parameter is used, typically found by accessing the TypeHelper or XSDHelper. An example of creating open content is found in the "Creating open content XML documents" on page 130.

All open content properties in getInstanceProperties() will have isSet(property) return true.

Property Indices

When a DataObject has multiple Properties, each of the Properties can be referenced by an index number, starting at 0 for the first Property.

The Property index used in getXXX(int property), is the position in the PropertyList returned by getInstanceProperties().

Using index parameter accessors for open content is **not** recommended if the data is being modified, unless the index is used in coordination with getInstanceProperties(). This is because the index of properties for open content in getInstanceProperties() can change, if the values of several open content properties are set and unset repeatedly.

The following example is acceptable because the index is used in coordination with getInstanceProperties(). Note that DataObjects are not synchronized so the user should not have updates going on at the same time as reads. This example shows a common pattern, looping through all instance properties and printing the property name and value:

```
const PropertyList& pl = myDo->getInstanceProperties();
for (int i=0; I < pl.size(); i++)
{
 const Property& p = pl[i];
  cout << p.getName() << "=" << myDo->getCString(i) << endl;
}
```

Names and alias names for Properties earlier in getInstanceProperties() take precedence over those with a higher index, meaning that open content Properties can have their name hidden by names defined in the Type's Properties since those Properties are at the beginning of the list. The order of precedence is the order in getInstanceProperties().

In the event of a duplicate name, the open content Property can be accessed through its alias name if that does not conflict with any names, or alias names, in the previous Properties.

Current State for a DataObject

The *current state* for a DataObject are all the values that distinguish it from a newly created object from the DataFactory, since newly created objects from a DataFactory have no properties set and no container. The current state for a DataObject are all the properties in getInstanceProperties() where isSet() returns true. The container and containment property are part of the state of the containing DataObject. This program prints the current state of the DataObject myDO.

```
DataObjectPtr dc = myDo->getContainingObject);
if (!dc)
{
cout << "There is no container" << endl;
}
else
{
cout << "Container of type " << dc->getType().getName();
}
const PropertyList& pl = myDo->getInstanceProperties();
for (int i=0; i< pl.size(); i++)
{
  const Property& p = pl[i];
  if (myDo->isSet(p))
```

```
  {
   if (p.getType().isDataType())
    {
      cout << p.getName() << "=" << myDo->getCString(i) << endl;
    }
    else
    {
      cout << p.getName() << "=" << myDo->getDataObject(i) <<
endl;
    }
  }
}
```

Introspection API on Class DataObject

---

unsigned int getPropertyIndex(const Property& p);

This returns a property index for this object from the property, or throws SDOPropertyNotFoundException.

---

const Property& getPropertyFromIndex(unsigned int index);

This returns a property for this object from the property index, or throws SDOPropertyNotFoundException.

PropertyList getInstanceProperties();

Returns the list of properties of this DataObject.The list may be empty. This will also return properties which are not part of the “getProperties()” list if the Type is an open type. This is a read-only list.

DataObjectPtr getContainer();

Returns a ref counting pointer to the containing DataObject. |If there is no container, then the boolean “if (!p)” will be true.

---

const Property& getContainmentProperty();

Return the Property from the containing DataObject whose value is this data object. Throws SDOPropertyNotFoundException if there is no container.

---

const Type& getType();

Returns the DataObject's type.

Type::Types getTypeEnum();

Returns the DataObject's type enum, which will be one of the enumerator Type::Types. See class Type.

PropertyList getProperties();

Returns the list of properties of this data objects type . This is a shorthand for getType().getProperties().

---

const ChangeSummary& getChangeSummary(const char path);*
const ChangeSummary& getChangeSummary(unsigned int propIndex);
const ChangeSummary& getChangeSummary(const Property& p);

The above will get the change summary for a particular DataObject in a graph. The list of changes in the ChangeSummary will reflect changes made whilst logging was switched on.
Change summaries are attached to DataObjects as a special Property, which is not visible to the getProperty API.

XPath

The following access methods reach data object values by locating the object relative to a known object using a subset of the XPath specification. This subset includes:

dob->getString(“propname”) - gets the value of “propname” of the data object dob.

dob->getString(“sub/propname”) – gets the value of the property “propname” of a data object whose value is held in the property “sub” of the data object dob.

dob->getString("subs[3]/propname") – subs is a many-valued data object property of the data object dob. This returns the value of the property "propname" of the third element of the list "subs".

dob->getString("subs.2/propname") – subs is a many-valued data object property of the data object dob. This is alternative syntax for getting value of the property "propname" of the third element of the list "subs".

dob->getString("subs[propname=fred]/altname") – subs is a many-valued data object property of the data object dob. This finds the element in the list subs for which the property "propname" has a value "fred", and returns the value of "altname" for that element.

dob->getDataObject(".."); - gets the container of data object dob.

The get/set API on Class DataObject

There are separate get and set methods for each primitive type and class which may be used, and these are overloaded such that the value can be located by path, property index or Property.

Each get/set call may throw either SDOPropertyNotFoundException or SDOTypeNotFoundException.

An SDOInvalidPathException is thrown if the path given does not specify a valid location of a data object.

All the APIs may throw SDOInvalidConversionException if the conversion is not supported.

Multi-valued properties are located by the getList/setList API.

Any primitive or datatype may also be set to null. This is a different state from 'unset' or 'set to zero'. When a primitive has been set to null, isSet() will return true, isNull() will return true, and getXXX() will return the default value for that type of property.

Multi-valued properties may not be set to null, neither may items within a multi-valued property.

DataObjectPtr getDataObject(const char path);*
DataObjectPtr getDataObject(unsigned int propertyIndex);

DataObjectPtr getDataObject(const Property& property);

Returns the value of a property of either this object or an object reachable from it, as identified by the specified path, index or property.
The parameter specifies either:

- the path to a valid object and property
- index of the property
- a Property reference.

Returns the value of the specified property as a ref counting pointer to a DataObject.
Throws an SDOInvalidConversionException if the object is primitive.
Throws an SDOInvalidPathException if the path does not represent a data object.

---

void setDataObject(const char path, DataObjectPtr value);*
void setDataObject(unsigned int propertyIndex, DataObjectPtr value);
void setDataObject(const Property& property, DataObjectPtr value);

This sets a property of either this object or an object reachable from it to the specified value.
The first parameter specifies either:

- The path to a valid object and property
- The index of the property
- The Property reference.

The second parameter is the new value for the property, expressed as a ref counting pointer to a data object.

---

bool getBoolean(const char path);*
bool getBoolean(unsigned int propertyIndex);
bool getBoolean(const Property& property);

This returns the value of a property identified by the specified path as a bool.
The parameter specifies either:

- The path to a valid object and property
- The index of the property
- A Property reference.

The return is the bool value of the specified property.

---

void setBoolean(const char path, bool value);*
void setBoolean(unsigned int propertyIndex, bool value);
void setBoolean(const Property& property, bool value);

Sets the value of a property of either this object or an object reachable from it to the specified bool value.
The parameter specifies either:

- The path to a valid object and property
- The index of the property
- A Property reference.

The second parameter is the new value for the property.

The get/set API above is duplicated for the following methods and return types:

char getByte(…)
void setByte(… char value)– a C++ char value returned from SDO Byte.

wchar_t getCharacter(…)
void setCharacter(…, wchar_t value)– wchar_t value returned from an SDO Character

short getShort(…)
void setShort(… short value)– short value returned from SDO Short

long getInteger(…)
void setInteger(… long value)– long value returned from SDO Integer

int64_t getLong(…)
void setLong(…int64_t value)– int64_t value returned from SDO Long

float getFloat(…)
void setFloat(…float value)– float value returned from SDO Float

long double getDouble(…)
void setDouble(…long double value)– long double value returned from SDO Double

SDODate getDate(…)
void setDate(…SDODate value)– SDODate instance returned from SDO Date

Note: An SDODate is a utility class which wraps a time_t. This avoids the platform specificity of time_t, and the possible clash of signature with methods requiring a ‘long’ parameter. The SDODate returned has a method getTime() which returns a time_t, or ascTime, which retutns a formatted C string.

const char* getCString(…)
void setString(… const char * value)– const char* value returned from any SDO type

Some conversions between types will result in loss of precision or overflow. These effects are detailed in the type conversions table later in this document.

Note: int64_t is defined by SDO as __int64 on windows platforms. On Linux platforms it is defined by the operating system.

Special handling of character buffers.

The SDO types String and Bytes may be retrieved as non null-terminated buffers of characters or wide characters.

There are two types of non null-terminated buffers available. These are wide characters (Type::String) and characters (Type::Bytes). The space to hold this type of buffer needs to be allocated by the application. There are convenience methods supplied to query the required length to allocate, and the query methods take a pointer to the allocated buffer as a parameter:

---

unsigned int getLength(const char path);*
unsigned int getLength(const Property& property);
unsigned int getLength(unsigned int propertyindex);

These return the current length of the data object of SDO type String (Type::String) or SDO type Bytes (Type::Bytes). If called on data objects of other types, this method will return the size of a buffer which would be needed to hold the value returned as a string.

---

unsigned int getString(const char path, wchar_t * buffer, unsigned int length);*
*unsigned int getString(const Property& property, wchar_t * buffer, unsigned int length);*
*unsigned int getString(unsigned int propertyindex, wchar_t * buffer, unsigned int length);*

These place the contents of the data objects property value into the wide char buffer, up to the length specified. The return value is the actual number of characters copied.

---

unsigned int getBytes(const char path, char * buffer, unsigned int length);*
*unsigned int getBytes(cosnt Property& property, char * buffer, unsigned int length);*
unsigned int getBytes(unsigned int propertyindex , char buffer, unsigned int length);*

These place the contents of the data objects property value into the char buffer, up to the length specified. The return value is the actual number of characters copied.

Note: The getter methods can also be used in place of the getLength methods if preferred. Passing a zero buffer and a zero length will result in the function returning the number of characters available, rather than the number of characters copied.

---

bool isSet(const char path);*
bool isSet(unsigned int propertyIndex);
bool isSet(const Property& property);

Returns whether a property of either this object or an object reachable from it is set.
The parameter specifies either:

- The path to a valid object and property
- The index of the property
- A Property reference.

---

void unSet(const char path);*
void unSet(unsigned int propertyIndex);
void unSet(const Property& property);

Un-sets a property of either this object or an object reachable from it.
The parameter specifies either:

- The path to a valid object and property
- The index of the property
- A Property reference.

---

void setNull(const char path);*
void setNull(unsigned int propertyIndex);
void setNull(const Property& property);

Sets a property of either this object or an object reachable from it to null.
The parameter specifies either:

- The path to a valid object and property
- The index of the property
- A Property reference.

---

bool isNull(const char path);*

void isNull(unsigned int propertyIndex);
void isNull(const Property& property);

Tests whether a property of either this object or an object reachable from it is null.
The parameter specifies either:

- The path to a valid object and property
- The index of the property
- A Property reference.

---

void detach();

Un-sets the property which contains this object, effectively detaching it from the object tree. A ref counting pointer to the object will still be valid after a detach, but the object will have no container. The objects properties remain set.

---

void clear();

Un-sets all the properties of this data object.

DataObject Accessor Exceptions

The following exceptions are thrown on DataObject accessors. These exceptions are all derived from SDORuntimeException. The contents of the exception are accessible through the SDORuntimeException API.
SDO specifies minimum functionality for implementations. An implementation may provide additional function so that valid results would be returned where this specification would produce an error, provided that the functionality is a strict superset and all valid uses of the SDO specification operate correctly.

getXXX(String path) and setXXX<T>(String path) only throw an SdoInvalidConversionException if it is impossible to convert between the actual and expected types.

Accessor methods may also throw SDOPropertyNotFoundException, SDOTypeNotFoundException or SDOInvalidPathException.

Methods may throw SDOUnsupportedOperationException, for example where an API returning a list is called on a single-valued property.

Open content DataObjects will not throw exceptions for accessing properties which are not set on the DataObject.

Validation of Facets and Constraints

XML elements can have facets, that is, restrictions. If the value set on a Property does not meet a facet or constraint, such as an XSD range restriction, the accessor may throw an IllegalArgumentException. However, implementations are not required to throw exceptions because it can be more practical to perform validation at a later time.

Validation that occurs during the execution of an accessor method is called *immediate* validation. Validation that is externally triggered is called *deferred* validation. In general, deferred validation is more efficient because checking is typically invoked once, after all values are set. Most constraints can only be enforced with deferred validation because more than a single property value is being validated. Underflow constraints (that is properties that must be assigned values for valid input to an application) are always deferred when building new DataObjects. SDO leaves it to implementations, applications, and services to determine when and how validation should be performed. Deferred validation is defined by services which perform validation on their input parameters, for example before the service makes updates to a database. Deferred validation does not occur through the DataObject APIs.

If an exception is thrown, no change to the DataObject takes place and therefore there is no change to any ChangeSummary.

Condition	Exception
For Types without open content (open=false), Property is not a member of getInstanceProperties() in get<T>(Property property) or get<T>(int propertyIndex). • getInstanceProperties().contains(property) == false • propertyIndex < 0 or >= getInstanceProperties().size() o Example: get(null) o Example: get(-1) o Example: isSet(property)	SdoIllegalArgumentException

Index out of range on a multi-valued Property (defined by the List interface) • index < 0 or >= getList(Property property).size() o Example: getList(employee).get(-1) o Example: getList(employee).get(1000) where there are less than 1000 values	SdoIndexOutOfRangeException
Modification of a read-only property • Example: set(employeeNumber, 123) where employeeNumber.isReadOnly() == true • Example: unset(employeeNumber) where employeeNumber.isReadOnly() == true • Example: getList(employees).remove(anEmployee) or • Example: anEmployee.detach() or • Example: anEmployee.delete() where employees.isReadOnly()==true and anEmployee.getContainmentProperty()==employees.	SdoUnsupportedOperationException
Cannot convert between value and requested Type • Example: getDate(property) where property.Type is float • Example: getList(property) where property.many == false and property.type.instanceClass is not List.	SdoInvalidConversionException
Mixing single-valued and multi-valued Property access • Example: getList(property) where property.many == false • Example: getInt(property) where property.many == true	SdoUnsupportedOperationException
Circular containment • Example: a.set("child", b); b.set("child", c); c.set("child", a) where child is a containment Property.	SdoIllegalArgumentException

DataGraph

A DataGraph is an optional envelope for a graph of DataObjects with a ChangeSummary.

To obtain the same functionality as the DataGraph with DataObjects alone, DataObjects may be defined using the SDO DataGraph XSD.

A ChangeSummary may be used directly with DataObjects as explained in the ChangeSummary section.

The DataGraph has methods to:

- return the root DataObject
- create a rootDataObject if one does not yet exist.
- return the change summary
- look up a type by uri and name .

DataGraph API

DataObjectPtr getRootObject();

Returns the root object of this data graph if the object exists . Otherwise returns a pointer to a zero value, such that the test (!p) will return true.

---

*DataObjectPtr createRootObject(const char * URI, const char * name);*

Creates and returns the root data object, using the factory specified in the data graph. May throw SDOTypeNotFoundException if the type is not known to the data factory.

DataObjectPtr createRootObject(const Type& t);

Creates and returns the root data object, using the factory specified in the data graph. May throw SDOTypeNotFoundException if the type is not known to the data factory.

ChangeSummaryPtr getChangeSummary();

Returns a pointer to the change summary for this datagraph, or zero if there is no change summary.

---

const Type& getType(const char uri, const char* name)*

Returns the type corresponding to the name and URI, according to the data factory in the data graph.

Creating DataGraphs

A DataGraph is created by a DAS, which returns either an empty DataGraph, or a DataGraph filled with DataObjects. An empty DataGraph can have a root assigned by the createRootObject() methods. However, if a previous root DataObject exists then an SdoUnsupportedOperationException is thrown.

The DAS is also responsible for the creation of the metadata (that is, the model) used by the DataObjects and DataGraph. For example, a DAS for XML data could construct the model from the XSD for the XML.

Modifying DataGraphs

In order to change a DataGraph a program needs to access the root DataObject, using the getRootObject() method. All other DataObjects in the tree are accessible by recursively traversing the containment references of the root DataObject.

Accessing Types

A Type can be accessed using getType(String uri, String typeName) or through the TypeHelper. This returns a Type with the appropriate URI and name. The convention for getType(), and all methods with a URI parameter, is that the URI is a logical name such as a targetNamespace.

The implementation of DataGraph and DataObject is responsible for accessing the physical resource that contains the requested metadata. The physical resource can be a local copy or a resource on a network.

The configuration information necessary to provide this logical to physical mapping, is via implementation-specific configuration files.

If metadata is unavailable, then an implementation-specific exception occurs.

ChangeSummary

A ChangeSummary provides access to change history information for the DataObjects in a data graph. This API is not part of the user API, but is provided for use by Data Access Service providers.

A change history covers any modifications that have been made to a data graph starting from the point when logging was activated. If logging is no longer active, the log includes only changes that were made up to the point when logging was deactivated. Otherwise, it includes all changes up to the point at which the ChangeSummary is being interrogated. Although change information

is only gathered when logging is on, you can query change information whether logging is on or off. All of the information returned is read-only.

This interface has methods that:

- Activate and deactivate logging.
- Restore a tree of DataObjects to the state it was in when logging began; and clear the log.
- Query the logging status.
- Get the DataGraph to which the ChangeSummary belongs.
- Get the ChangeSummary's root DataObject.
- Get the changed DataObjects.
- Indicate whether a DataObject has been created, deleted or changed.
- Get the container DataObject at the point when logging began.
- Get a DataObject's containment Property at the point when logging began.
- Get a DataObject's Sequence at the point when logging began.
- Get a specific old value.
- Get a List of old values.

Scope

The scope of a ChangeSummary is defined as the containment tree of DataObjects from the ChangeSummary root. The ChangeSummary root is the DataObject from which all changes are tracked. The ChangeSummary root is retuned by getRootObject(). This object is one of the following:

- The DataObject with the ChangeSummary as a value.
- The root DataObject of a DataGraph.

Old Values

A List of old values can be retrieved using the getOldValues(DataObject dataObject) method. The order of old values returned is implementation dependent. For a deleted DataObject, the old values List contains all the properties of the DataObject. For a DataObject that has been modified, the old values List consists of the modified properties only. For a DataObject that has not been deleted or modified, the List of old values is empty.

Old values are expressed as Setting objects Each Setting has a Property and a value, along with a flag to indicate whether or not the Property is set.

getOldValue(DataObject dataObject, Property property) returns a Setting for the specified Property, if the DataObject was deleted or modified. Otherwise, it returns null. If the setting.isSet() of the old value is false, the old value does not have meaning.

Sequenced DataObject

getOldSequence(DataObject dataObject) returns the entire value of a DataObject's Sequence, at the point when logging began. This return value can be null. If DataObject.getSequence() returns null then getOldSequence(DataObject dataObject) will return null.

Serialization and Deserialization

When a ChangeSummary is deserialized, the logging state will be on if a <changeSummary> element is present in the XML unless the changeSummary marks logging as off. A serializer must produce a <changeSummary> element in the XML if either of the following conditions applies:

1. Changes have been logged (getChangedDataObjects().size() > 0).
2. No changes have been logged **but** isLogging() is true at the time of serialization. In this case, an empty <changeSummary/> or <changeSummary logging="true"/> element must be produced.

The state of logging is recorded in the logging attribute of the changeSummary element.

The serialization of a ChangeSummary includes enough information to reconstruct the original information of the DataObjects, at the point when logging was turned on. The create attribute labels DataObjects currently in the data graph that were not present when logging started, and the delete attribute labels objects contained in the change summary that are no longer in the data graph. Labels are either IDs, if available, or sdo path expressions.

The contents of a ChangeSummary element are either deep copies of the objects at the point they were deleted, or a prototype of an object that has had only data type changes, with values for the properties that have changed value.

Associating ChangeSummaries with DataObjects

There are two possible ways to associate DataObjects and ChangeSummaries:

1. DataGraphs can get a ChangeSummary using the getChangeSummary() method.
   - This is used when a ChangeSummary is external to the DataObject tree. The ChangeSummary tracks changes on the tree of DataObjects starting with the root DataObject available through DataGraph's getRootObject().
2. The Type of a DataObject can include a Property for containing a ChangeSummary.
   - This is used when a ChangeSummary is part of a DataObject tree, for example when a root DataObject is a message header that contains both a message body of DataObjects and a ChangeSummary. The ChangeSummary tracks changes on the tree of DataObjects starting with the DataObject that contains the ChangeSummary.

ChangeSummary API

---

bool isLogging();

Returns true if the ChangeSummary is currently logging changes.

---

void beginLogging()

beginLogging() clears the ChangeSummary's List of changed DataObjects and starts change logging. endLogging() stops change logging. undoChanges() restores the tree of DataObjects to its state when logging began. undoChanges() also clears the log, but does not affect isLogging().

NOTE: The beginLogging(), endLogging() and undoChanges() methods are intended primarily for the use of service implementations since services define how the processing of a ChangeSummary relates to external resources. Making changes that are not captured in the ChangeSummary may cause services that drive updates from a ChangeSummary to act on incomplete information.

---

void endLogging()

Stops the logging process for this change summary.

---

void undoChanges()

Resets the data graph back to its state when logging was last switched on.

---

const ChangedDataObjectList getChangedDataObjects();

Gets the list of DataObjects whose properties or value have been modified, added or deleted since logging was last switched on. This method returns a ChangedDataObjectList, which can queried for size, and each element addressed using the index notation the_list[index].

---

bool isModified(DataObjectPtr dataobject);

This method returns true if the data object has been modified since logging was switched on

---

bool isDeleted(DataObjectPtr dataobject);

This method returns true if the data object has been deleted since logging was switched on. In this case the data object pointer is not guaranteed to be a valid data object.

---

bool isCreated(DataObjectPtr dataobject);

This method returns true if the data object has been created since logging was switched on.

---

const Setting& getOldValue(DataObjectPtr dataobject, const Property & property)

getOldValue(DataObjectPtr dataObject, const Property& property) returns a Setting for the specified Property, if the DataObject was deleted or modified. Otherwise, it returns an empty Setting. If the setting.isSet() of the old value is false, the old value does not have meaning.

SettingList& getOldValues(DataObjectPtr dataObject);

A List of old values can be retrieved using the getOldValues(DataObjectPtr dataObject) method. The order of old values returned is implementation dependent. For a deleted DataObject, the old values List contains all the properties of the DataObject. For a DataObject that has been modified, the old values List consists of the modified properties only. For a DataObject that has not been deleted or modified, the List of old values is empty.

Old values are expressed as a list of Setting objects Each Setting has a Property and a value, along with a flag to indicate whether or not the Property is set.

DataObjectPtr getOldContainer(DataObjectPtr dataobject);

Returns the old containing data object of a data object which has been deleted.

---

const Property& getOldContainmentProperty(DataObjectPtr dataobject);

Returns the old containment property of a data object which has been deleted. This will throw an exception if called when getOldContainer has returned a pointer for which “if (!p) is true. (I.E There was no containing object)..

```
SequencePtr getOldSequence(DataObjectPtr dataObject);
```

getOldSequence(DataObjectPtr dataObject) returns the entire value of a DataObject’s Sequence, at the point when logging began. This return value can be null. If DataObject.getSequence() returns null then getOldSequence(DataObject dataObject) will return null.

Sequence

A Sequence is an ordered collection of settings. Each entry in a Sequence has an index.

The key point about a Sequence is that the order of settings is preserved, even across different properties. So, if Property A is updated, then Property B is updated and finally Property A is updated again, a Sequence will reflect this.

Each setting is a property and a value. There are shortcuts for using text with the SDO text Property.

Unstructured Text

Unstructured text can be added to a Sequence using the SDO text Property. The add(const char * text) method adds a new entry, with the SDO text Property, to the end of the Sequence. The add(int index, const char * text) method adds a new entry, with the SDO text Property, at the given index.

Using Sequences

Sequences are used when dealing with semi-structured business data, for example mixed text XML elements. Suppose that a Sequence has two many-valued properties, say “numbers” (a property of type int) and “letters” (a property of type String). Also, suppose that the Sequence is initialized as follows:

1. The value 1 is added to the numbers property.
2. The String “annotation text” is added to the Sequence.
3. The value “A” is added to the letters property
4. The value 2 is added to the numbers property.
5. The value “B” is added to the letters property.

At the end of this initialization, the Sequence will contain the settings:

{<numbers, 1>, <text, ”annotation text”>, <letters, ”A”>, <numbers, 2>, <letters, ”B”>}

The numbers property will be set to {1, 2} and the letters property will be set to {“A”, ”B”}, but the order of the settings across numbers and letters will not be available through accessors other than the sequence.

Comparing Sequences with DataObjects

The way in which a DataObject keeps track of the order of properties and values is quite different from the way in which a Sequence keeps track of the order.

The order in which different properties are added to a DataObject is not preserved. In the case of a many-valued Property, the order in which different values are added to that one Property is preserved, but when values are added to two different Properties, there is no way of knowing which Property was set first. In a Sequence, the order of the settings across properties is preserved.

The same properties that appear in a Sequence are also available through a DataObject, but without preserving the order across properties.

Note that if a DataObject's Type is a sequenced type (that is, if getType().isSequenced() is true) then a DataObject will have a Sequence.

The Sequence API

---

unsigned int size();

Returns the number of elements in the sequence

---

bool isText(unsigned int index);

Returns true if the element is free text, or false if the element is a property setting.

---

const Property& getProperty(unsigned int index);

An element in a sequence represents a setting of a value of a data objects property. This setting may be a setting of as single valued property, or an addition to a list of values for a many-valued property, or a free text element with no associated property. The above API establishes which property is at a specified element in the sequence. An exception is thrown if this is called on a free text element.

---

unsigned int getListIndex(unsigned int index);

If the setting represents a modification to an element in a list of values, this method returns the index at which the value was set. The method will return zero if called on a setting which represents a modification to a single-valued property

---

unsigned int getIndex(const Property& p, unsigned int pindex=0);

This method returns the index in the setting list at which this property was modified. If the property is many-valued, then the index in the list of values pindex is used to find the correct setting.

---

unsigned int getIndex(const char propName, unsigned int pindex=0);*

This method returns the index in the setting list at which the named property was modified. If the property is many-valued, then the index in the list of values pindex is used to find the correct setting. An element in a sequence represents a setting of a value of a data objects property. This setting may be a setting of as single valued property, or an addition to a list of values for a many-valued property, or a free text element with no associated property.

If the value returned from isText() is false, then the setting is a single or many-valued property. If property.isMany() is true, then the getListIndex call will return the index within the many-valued property at which the setting took placed.

The sequence element which contains a specific property/index pair can be established from the getIndex methods. If the second value is absent, the call assumes you are referring to a single-valued property.

Type::Types getTypeEnum(unsigned int index);

Returns the enum value from the list of primitive types. This allows easy switching on Type.

---

void addBoolean (const char PropertyName, bool value);*
void addBoolean(const Property& Prop, bool value);
void addBoolean (unsigned int PropertyIndex,bool value);
void addBoolean (unsigned int index, const char PropertyName, bool value);*
void addBoolean(unsigned int index, const Property& Prop, bool value);
void addBoolean (unsigned int index, unsigned int PropertyIndex,bool value);

The value passed as a bool is used to set the property specified by the index, name or property reference.

Adding a setting for a single-valued property to a sequence which already contains that setting will result in an SDOUnsupportedOperationException.

Adding a setting for a multi-valued property to a sequence which already contains a setting for that property will add the element to the property, and add an element to the sequence. Sequence items have an API to discover the index of the value within the many-valued property.

The add method appends an element at the end of the setting list.

Each add method has an override taking an unsigned int index. This inserts the value before the given index in the sequence. If the index is greater than the sequence length, then the item is appended.

The above add method is duplicated for each primitive (and dataobject):

void addByte() – char value
void addCharacter() – wchar_t value
void addBytes() – char value, unsigned int length*
void addString() – wchar_t value, unsigned int length*
void addCString() – char value*
void addShort()– short value
void addInteger()– long value
void addLong()– int64_t value
void addFloat()– float value
void addDouble()– long double value
void addDate() – SDODate value
void addDataObject() – DataObjectPtr value

---

void addText(const char value);*
void addText(unsigned int index, const char value);*

A sequence may contain items of text between the property elements. These can be added by the addText methods. These methods add an element to the sequence which has no corresponding property setting. The API to query the setting will result zero for the property pointer for these elements.

---

bool isText(unsigned int index)

This method returns true if the element is text, or false if the element is a property setting.sets the

text in a text element. The element must be of free text type so it is advisable to use the isText method before this call.

---

void setText(unsigned int index, const char value)*

This method sets the text in a text element. The element must be of free text type so it is advisable to use the isText method before this call.

---

void remove (unsigned int index);

Removes an element from the sequence, and unsets the corresponding property in the DataObject.

---

void move(unsigned int toIndex, unsigned int fromIndex);

This method modifies the position of the item in the sequence, without altering the value of the property in the DataObject.

DataObjectPtr getDataObject(unsigned int index);

This method returns the DataObject found at this location in the sequence. This will throw an SDOIndexOutOfRangeException if the index is not within the sequence.

As with the data object API, there is a get method for each type of primitive.

---

bool getBooleanValue(unsigned int index);
char getByteValue(unsigned int index);
wchar_t getCharacterValue (unsigned int index);
unsigned int getBytesValue(unsigned int index, char buffer, unsigned int max);*
unsigned int getStringValue (unsigned int index, wchar_t buffer, unsigned int max);*
short getShortValue (unsigned int index);
long getIntegerValue (unsigned int index);
int64_t getLongValue (unsigned int index);
float getFloatValue (unsigned int index);
long double getDoubleValue (unsigned int index);
SDODate getDateValue(unsigned int index);
const char getCStringValue (unsigned int index);*

---

unsigned int getLength(unsigned int index);

If the element is of type String (Type::String) or Bytes (Type::Bytes), this method returns the length of the buffer required to hold the value. Otherwise the method returns the maximum size of a buffer required to hold a value of this type as a string.

---

void setDataObjectValue(unsigned int index, DataObjectPtr value);
void setBooleanValue(unsigned int index, bool value);
void setByteValue(unsigned int index, char value);
void setCharacterValue(unsigned int index, wchar_t value);
void setBytesValue(unsigned int index, char value, unsigned int length);*
void setStringValue(unsigned int index, wchar_t value, unsigned int length);*
void setShortValue(unsigned int index, short value);
void setIntegerValue(unsigned int index,long value);
void setLongValue(unsigned int index,int64_t value);
void setFloatValue(unsigned int index, float value);
void setDoubleValue(unsigned int index, long double value);
void setDateValue(unsigned int index, SDODate value);
void setCStringValue(unsigned int index, const char value);*

The value of a property at a location in the sequence may be set using one of the set APIs. These will throw an SDOIndexOutOfRangeException if the index is not within the sequence.

Type

The Type interface represents a common view of the model of a DataObject, or of a data type.

The concept of a data type is shared by most programming languages and data modeling languages; and SDO Types can be compared with other data types. An SDO Type has a set of Property objects, unless it represents a simple data type.

Mapping SDO Types to Programming and Data Modeling Languages

Java, C++, UML or EMOF Class

- Class can be represented by an SDO Type.
- Each field of the Class can be represented by an SDO Property.

XML Schema

- Complex and simple types can be represented by SDO Types.
- Elements and attributes can be represented by SDO Properties.

C Struct

- C Struct can be represented by an SDO Type
- Each field of the Struct can be represented by an SDO Property.

Relational database

- Table can be represented by an SDO Type.
- Column can be represented by an SDO Property.

All of these domains share certain concepts, a small subset of which is represented in the SDO Type and Property interfaces. These interfaces are useful for DataObject programmers who need to introspect the shape or nature of data at runtime.

More complete metamodel APIs (for example, XML Schema or EMOF) representing all the information of a particular domain are outside the scope of this specification.

Type Contents

A Type will always have:

- Name - A String that must be unique among the Types that belong to the same URI.
- URI - The logical URI of a package or a target namespace, depending upon your perspective.
- Boolean fields indicating if the type is open, abstract, sequenced, or a data type.

A Type can have:

- Properties - a list of Property objects defined by this Type. Types corresponding to simple data types define no properties.
- Aliases - Strings containing additional names. Alias Names must be unique within a URI. All methods that operate on a Type by name also accept alias names. For example, a Type might be assigned an alias name for the domains it is used in: an XML Schema name "PurchaseOrderType", a Java name "PurchaseOrder" and a database table name "PRCHORDR".

Name Uniqueness

Type names and Type alias names are all unique within a URI. Property names and Property alias names are all unique within a Type and any base Types.

SDO Data Types

SDO defines Types for the common data types supported in SDO, enabling more consistency in defining the Types and Properties used by services. Refer to "Standard SDO Types" on page 77 for more details.

Inheritance

Type supports inheritance by allowing a base type. When inheritance is used, the properties returned by getProperties() of the subtype will be those of the supertype, followed by those of the subtype.

The Type API

---

const char getName() const;*

Returns the name of the type.

---

const char getURI() const;*

Returns the namespace URI of the type.

---

const char getAlias(unsigned int index = 0);*

Return the alias of this Type from the 'index'th element of its list of aliases.

Note that it is only possible to add aliases to a Type using the SPI, prior to creation of the first instance of a data object.

---

unsigned int getAliasCount() const = 0;

Return the number of aliases that this Type has.

PropertyList getProperties() const;

Returns the list of the properties belonging to this type.

---

const Property& getProperty(const char propertyName) const;*

Returns the property with this name. This method will throw SDOPropertyNotFoundException if the property does not exist.

---

const Property& getProperty(unsigned int index) const ;

Returns the property at this index. This method will throw SDOPropertyNotFoundException if the property does not exist.

---

unsigned int getPropertyIndex(const char propertyName) const;*

Returns the property index for the property with this name. This method will throw SDOPropertyNotFoundException if the property does not exist.

---

bool isDataObjectType() const ;

Indicates if this Type specifies a DataObject. Returns true if this Type specifies a DataObject, false if it is a primitive.

---

bool isDataType() const ;

Indicates if this Type specifies a Primitive. Returns true if this Type specifies a primitive, otherwise false.

---

bool isSequencedType() const ;

Indicates if this Type specifies a sequenced DataObject. For more information on sequences, see the Sequence API.

---

bool isOpenType() const ;

Indicates if this Type allows any form of open content. If isOpenType returns false then dataObject.getInstanceProperties() must be the same as dataObject.getType() .getProperties().

Type::Types getTypeEnum() const ;

Returns the type enum for this type, which may be any of the following values:

Type::OtherTypes – any unrecognized type,
Type::BigDecimalType,
Type::BigIntegerType,
Type::BooleanType,
Type::ByteType,

Type::BytesType,
Type::CharacterType,
Type::StringType,
Type::DateType,
Type::DoubleType,
Type::FloatType,
Type::IntegerType,
Type::LongType,
Type::ShortType,
Type::UriType,
Type::DataObjectType,
Type::ChangeSummaryType,
Type::TextType.

Property

A DataObject is made up of Property values.

A Property has:

- Name - a String that is typically unique among the Properties that belong to the DataObject.
- Type - the Type of this Property. A Property whose Type is for DataObjects is sometimes called a reference; otherwise it is called an attribute.
- Containment - whether the property is a containment property. A property with containment true is called a ***containment property***.
- Many - whether the property is single-valued or many-valued.
- ReadOnly - whether the property may be modified through the DataObject or generated API.
- Alias names - alternative names that must be unique within the Type. A Property might be assigned an alias name for the domains it is used in, such as an XMLSchema name "firstName", a C++ name "first_name", and a database column name, "FRSTNAME". All Property names and all alias names for Properties must be unique for all Properties in Type.getProperties().
- Default value.
- Numeric index within the Property's Type.

Property API

---

const char getName() const;*

Returns the name of the property.

---

const Type& getType() const;

Returns the type of the property.

---

const char getAlias(unsigned int index = 0)*

Return the alias of this property from the ‘index’th element of the list of aliases.;

---

unsigned int getAliasCount();

Return the number of aliases which belong to this property.

Note that it is only possible to add aliases to a property via the SPI before the first data object instance is created.

Type::Types getTypeEnum();

Return the type of this property as an enumeration value. This is an aid to providing switching on Type. See Type::getTypes.

---

bool isMany() const;

Returns whether the property is many-valued. Many-valued properties can be accessed via the getList interface.

---

bool isContainment() const;

Returns whether the property is containment, i.e., whether it represents by-value composition.

---

const Type& getContainingType() const;

Returns the Type that contains this property.

---

bool isReference() const ;

Returns true if the property is a reference to a value within another data object type. Returns false if the property is a value contained in a data object of this type.

---

bool isReadOnly() const;

Returns true if values for this Property cannot be modified using the SDO APIs. Attempting to set values of read-only properties will provoke an SDOUnsupportedOperationException.
Values of the property may change due to other factors such as services operating on data objects.

Property getOpposite() const;*

Get the property which is the opposite of this in a bi-directional relationship. Zero is returned if there is no opposite.

Containment

In the case of a reference, a Property may be either a containment or non-containment reference. In EMOF, the term containment reference is called composite. In XML, elements with complex types are mapped to containment properties.

A Property with containment true is called a containment property. Containment properties show the parent-child relationships in a tree of DataObjects.

Read-Only Properties

Read-Only Properties **cannot** be modified using the SDO APIs. When DataObject.delete() is invoked, read-only Properties are not changed. Any attempt to alter read-only Properties using DataObject.set(Property property, Object value) or unset() results in an exception.

Read-Only Properties **can** be modified by a service using implementation-specific means. For Example, suppose a relational database service returns a DataObject in which the customerName Property is marked read-only. If the DataObject is passed back to the service, the value of the customerName could be updated by the service to the current value in the database.

PropertyList, TypeList and DataObjectList

Lists provide a containing structure for groups of objects.

PropertyList

const Property& operator[] (unsigned int pos) const;

Returns the Property reference at the index specified in the list.

---

unsigned int size ();

Returns the number of elements in the list.

TypeList

---

const Type& operator[] (unsigned int pos) const;

Returns the Type reference at the index specified in the list.

---

unsigned int size ();

Returns the number of elements in the list.

DataObjectList

DataObjectPtr operator[] (unsigned int pos);

Returns the DataObject element at the index specified in the list as a ref counting pointer.

---

const DataObjectPtr operator[] (unsigned int pos) const;

Returns the const DataObject element at the index specified in the list.

---

unsigned int size ();

Returns the number of elements in the list.

---

void insert (unsigned int index, DataObjectPtr value);

Inserts a DataObject at the specified index, and moves elements after the index up one.
If the index is above the current size of the list, the item will be appended.

---

void append (DataObjectPtr value);

Appends the DataObject to the end of the list.

---

unsigned int getLength(unsigned int index);

Returns the length of the element at index ‘index’. This uses the DataObject getLength() and is intended for use with Strings and Bytes objects to find the required length of a buffer to allocate for a getString ot getBytes call.

Data object lists support all of the APIs found on dataObject, so these can be applied to the insert, append and [] operators of DataObjectList.

For example:
dolist->insert(index, bool);
dolist->append(short);
dolist->setBoolean(index, bool b);

SDORuntime

The SDO runtime class contains static methods to access the version information for the library.

---

unsigned int SDORuntime:: getMajor();

This returns the major version number of the runtime

---

unsigned int SDORuntime:: getMinor();

This returns the minor version number of the runtime

---

unsigned int SDORuntime:: getFix();

This returns the fix level of the runtime

---

const char SDORuntime:: getVersion();*

This returns a string representing the major version, minor version and fix leve l,

---

DataFactory

A DataFactory is a helper for the creation of DataObjects. The created DataObjects are not connected to any other DataObjects. Only Types with DataType false and abstract false may be created.

DataFactory API

Before a DataFactory can be used to create a DataObject, it must be populated with descriptions of the allowable Types and their corresponding Properties. The tree of Types is set into the DataFactory using the addType and addPropertyToType APIs, and then the tree is validated for circular dependencies and invalid base types when the first create of a DataObject is requested. The DataFactory, when first created, already contains definitions of the SDO primitive types such as String and Integer.

DataFactoryPtr getDataFactory();

This method returns the default data factory, which contains an implementation specific list of Types.

DataObjectPtr create(const char namespace, const char* name);*
DataObjectPtr create(const Type& t);

This creates a DataObject. The creation and setting APIs of this data object can then be used to create further objects in the graph.

---

const Type& getType(const char uri, const char* inTypeName);*

Returns a Type corresponding to the URI and name, or throws an exception if the type is not valid in this DataFactory.

TypeList getTypes();

Returns a list of types which are valid in this DataFactory.

---

void addType(const char uri, const char* inTypeName,bool isSequenced = false, bool isOpen = false, bool isDataType = false, bool isAbstract = false) ;*

Adds a type to this data factory.

```
void addPropertyToType(const char* uri,const char* inTypeName,
    const char* propname,
    const char* propTypeUri,
    const char* propTypeName,
    bool isMany  ,
    bool  isReadOnly ,
    bool isContainment ) ;
```

Adds a property to the specified Type. Specifying whether the new property will be many-valued, read-only or containment.
This method has overrides specifying the Type being added to, and the Type of the property as const Type& rather than URI/name pairs.

```
void setBaseType( const Type& type,const Type& base);
void setBaseType( const char* typeuri,const char* typenam,
                  const char* baseuri,const char* basename) ;
```

A Type may inherit properties from a base type. These base types are declared by setting the base type once the type has been added to the DataFactory. The properties of the base type appear first in the property list of the subtype.

```
void setAlias(const char* typeuri,const char* typenam,const char* alias) = 0;
void setAlias(const char* typeuri,const char* typname,
              const char* propname,const char* alias) ;
```

A type or property may have many alias names. Each name set by this method becomes an alias

for the specified Type or Property.

---

void setDefault(const Type& t, const char propname, bool b);*

Each property may have a default value, this overrides the default applied to all properties of that Type. This method has overrides for each primitive type (Integer, Boolean, Short...)

---

void setOpposite(const Type& t, const char propname,*
const Type& t2, const char oppositename);*

A property may have an opposite property , This is set by this API call.

Defining SDO Types Dynamically

It is possible to define new SDO Types dynamically using TypeHelper. For example, to define a new Customer Type you could use the TypeHelper as follows:

```
DASDataFactory myDf = DASDataFactory::getDataFactory();
TypeHelper& types = TypeHelper( myDf );

const Type& intType = types.getType("commonj.sdo", "Integer");
const Type& stringType = types.getType("commonj.sdo", "String");

myDf->addType("http://example.com/customer","Customer");
myDf->addPropertyToType("http://example.com/customer","Customer",
                        "custNum", intType);
myDf->addPropertyToType("http://example.com/customer","Customer",
                        "firstName", stringType);
myDf->addPropertyToType("http://example.com/customer","Customer",
                        "lastName", stringType);
```

Using SDO Dynamic Types

To use the dynamically created Customer Type you could do as follows:

```
DASDataFactory myDf = DASDataFactory::getDataFactory();
```

```
DataObjectPtr customer1 = myDf.create("http://example.com/customer",
"Customer");
customer1.setInteger("custNum", 1);
customer1.setCString("firstName", "John");
customer1.setCString("lastName", "Adams");

DataObjectPtr customer2 = factory.create("http://example.com/customer",
"Customer");
customer2.setInteger("custNum", 2);
customer2.setCString("firstName", "Jeremy");
customer2.setCString("lastName", "Pavick");
```

CopyHelper

A CopyHelper creates copies of DataObjects.
A copy helper contains only static methods to allow deep or shallow copying of DataObjects.

CopyHelper API

DataObjectPtr copyShallow(DataObjectPtr dob);

copyShallow(DataObjectPtr dataObject) creates a new DataObject with the same values as the source dataObject, for each Property where property.type.dataType is true.
If the source's property.type.dataType is false, then that property is unset in the copied DataObject. Read-only properties are copied.
The resulting DataObject will initially have no containing data object.
If a ChangeSummary is part of the source DataObject then the copy has a new, empty ChangeSummary. The logging state of the new ChangeSummary is the same as the source ChangeSummary.

DataObjectPtr copy(DataObjectPtr dob);

copy(DataObject dataObject) creates a deep copy of the DataObject tree, that is it copies the dataObject and all its contained DataObjects recursively.

For each Property where property.type.dataType is true, the values of the Properties are copied as in the shallow copy. Read-only properties are copied.

For each Property where property.type.dataType is false, the value is copied if it is a DataObject contained by the source dataObject.

If a DataObject is outside the DataObject tree and the property is bidirectional, then the DataObject is skipped. If a DataObject is outside the DataObject tree and the property is unidirectional, then the same DataObject is referenced.

If a ChangeSummary is part of the copy tree then the new ChangeSummary refers to objects in the new DataObject tree. The logging state is the same as for the source ChangeSummary.

EqualityHelper

An EqualityHelper compares DataObjects to decide if they are equal.

The EqualityHelper contains only static methods to allow comparison at a single level (equalShallow) , or a comparison of an entire data graph down from a DataObject (equal).

EqualityHelper API

---

bool equalShallow(DataObjectPtr d1, DataObjectPtr d2);

> equalShallow(DataObjectPtr dataObject1, DataObjectPtr dataObject2) returns true if two DataObjects have the same Type, and all their compared DataType Properties are equal.

---

bool equal(DataObjectPtr d1, DataObjectPtr d2);

> equal(DataObjectPtr dataObject1, DataObjectPtr dataObject2) returns true if two DataObjects are equalShallow(), all their compared Properties are equal, and all reachable DataObjects in their graphs (excluding containers) are equal.

XMLHelper

An XMLHelper converts XML streams to and from graphs of DataObjects.

An XML Helper is accessed via a static method of the class HelperProvider, and is connected to the DataFactory by passing the factory as a creation parameter, thus:

```
XMLHelperPtr myXMLHelper = HelperProvider::getXMLHelper(myDataFactory);
```

XMLHelper can be used with or without an XSD. All closed trees of DataObjects are supported, whether or not an XSD was specified. However, the XML will use an XSD if one is used to define the DataObjects.

XMLHelper supports the case where a DataObjects's Types and Properties did not originate in an XSD. It does this by writing XML documents that follow the Generation of XSDs portion of this specification.

XMLHelper API

XMLDocumentPtr loadFile(const char xmlFile,const char* targetNamespaceURI=0) ;*

XMLDocumentPtr load(std::istream& inXml,const char targetNamespaceURI=0) ;*

XMLDocumentPtr load(const char inXml,const char* targetNamespaceURI=0) ;*

---

void save(XMLDocumentPtr doc, const char xmlFile);*

void save(
DataObjectPtr dataObject,
const char rootElementURI,*
const char rootElementName,*
const char xmlFile);*

Serializes the datagraph to the XML file.

---

void save(XMLDocumentPtr doc, std::ostream& outXml) ;

void save(
DataObjectPtr dataObject,
const char rootElementURI,*

const char rootElementName,*
std::ostream& outXml) ;

Serializes the datagraph to an output stream.

---

char save(XMLDocumentPtr doc);*

char save(*
DataObjectPtr dataObject,
const char rootElementURI,*
const char rootElementName) ;*

Serializes the datagraph to a string,

XMLDocumentPtr createDocument(
DataObjectPtr dataObject,
const char rootElementURI,*
const char rootElementName);*

Loading and Saving XML Documents

The XMLHelper and XMLDocument do not change the state of the input DataObject and ignore any containers. After load, the root DataObject created does not have a containing DataObject.

When loading XML documents, typically the Types and Properties are already defined, for example from an XSD. If there are no definitions, the XML without Schema to XSD is used. In some situations, the definitions of the Types and Properties have changed relative to the software that has originally written the document, often called *schema evolution*. SDO does not directly address schema evolution, which is an issue broader than SDO, but the general guideline is to use the same URI for compatible XML documents and different URIs for incompatible XML documents.

XML Schemas

Often, it is desirable to validate XML documents with an XSD. To ensure validation, the root element name and URI must correspond to a global element name and target namespace in an XSD.

If an XSD is not being used, for example when the schema types were created dynamically with TypeHelper, then the following is suggested. Choose element names and URIs from the global elements that would be produced by the Generation of XSDs. This improves integration with software that does make use of XSDs.

To enable XML support for DataObjects when XSDs are not used, the following conventions apply to root elements. Although the use of a valid global element is not enforced, it is encouraged whenever an XSD is available.

- When saving the root element, an xsi:type may always be written in the XML to record the root DataObject's Type. If the rootElementURI and rootElementName correspond to a valid global element for that DataObject's Type, then an implementation may suppress the xsi:type.
- When loading the root element, if an xsi:type declaration is found, it is used as the type of the root DataObject. Unless XSD validation is being performed, it is not an error if the rootElementURI and rootElementName do not correspond to a valid global element.
- The root element "commonj.sdo", "dataObject" may be used with any DataObject if xsi:type is also written for the actual DataObject's Type.

Creating DataObjects from XML

Using XMLHelper it is easy to convert between XML and DataObjects. The following example shows how to get a DataObject from XML:

```
char poXML[1000];

strcpy(poXML, "<purchaseOrder orderDate='1999-10-20'>");
strcat(poXML, "  <shipTo country='US'>");
strcat(poXML,  "    <name>Alice Smith</name>");
strcat(poXML,  "    <street>123 Maple Street</street>");
strcat(poXML,  "    <city>Mill Valley</city>");
strcat(poXML,  "    <state>PA</state>");
strcat(poXML,  "    <zip>90952</zip>");
strcat(poXML,  "  </shipTo>");
strcat(poXML,  "</purchaseOrder>");

XMLDocumentPtr xdo = xmlHelper->load(poXML);
```

Creating DataObjects from XML documents

It is possible to convert to and from XML documents to build DataObject trees, which is useful when assembling DataObjects from several data sources. For example, suppose the global elements for shipTo and billTo were declared in the PurchaseOrder XSD:

```
<element name="shipTo" type="USAddress"/>
<element name="billTo" type="USAddress"/>
```

To create the shipTo DataObject from XML:

```
char*  shipToXML[1000];

strcpy(shipToXML, "<shipTo country='US'>");
strcat(shipToXML,  "  <name>Alice Smith</name>");
strcat(shipToXML,  "  <street>123 Maple Street</street>");
strcat(shipToXML,  "  <city>Mill Valley</city>");
strcat(shipToXML,  "  <state>PA</state>");
strcat(shipToXML,  "  <zip>90952</zip>");
strcat(shipToXML,  "</shipTo>");

DataObjectPtr shipTo = xmlHelper->load(shipToXML)->getRootObject();
purchaseOrder.setDataObject("shipTo", shipTo);
```

To convert the billTo DataObject to XML:

```
const char* billToXML = xmlHelper->save(billTo, null, "billTo");
cout << billToXML << endl;
```

This produces:

```
<?xml version="1.0" encoding="UTF-8"?>
<billTo country="US">
  <name>Robert Smith</name>
  <street>8 Oak Avenue</street>
  <city>Mill Valley</city>
  <zip>95819</zip>
</billTo>
```

Creating XML without an XSD

XMLHelper can be used without an XSD. In the TypeHelper Customer example, a Customer Type was defined dynamically without an XSD and then customer1 was created. To save customer1 to XML:

```
xmlHelper->save(customer1, "http://example.com/customer", "customer", stream);
```

This produces the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<customer xsi:type="Customer" custNum="1" firstName="John" lastName="Adams"
  xmlns="http://example.com/customer"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
```

An example of using the XML Helper in a simple web services client is found in the section Web Services Client using XMLHelper.

XMLDocument

An XMLDocument represents an XML Document containing a graph of DataObjects.

XMLHelper creates and serializes XMLDocument objects. An XMLDocument enables a program to access parts of an XML Document.

XMLDocuments do not change the state of any input DataObjects and ignore any containers.

XMLDocument API

DataObjectPtr getRootDataObject() const;

---

const char getRootElementURI() const ;*

---

const char getRootElementName() const ;*

---

const char getEncoding() const ;*

---

void setEncoding(const char encoding) ;*

---

bool getXMLDeclaration() const ;

---

void setXMLDeclaration(bool xmlDeclaration) ;

---

const char getXMLVersion() const ;*

---

void setXMLVersion(const char xmlVersion) ;*

---

const char getSchemaLocation() const ;*

---

void setSchemaLocation(const char schemaLocation) ;*

---

const char getNoNamespaceSchemaLocation() const ;*

---

void setNoNamespaceSchemaLocation(const char noNamespaceSchemaLocation) ;*

Example XMLDocument

Using this XML Schema fragment:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
  <xsd:complexType name="PurchaseOrderType">
```

and the following example XMLDocument fragment:

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
```

After loading this XMLDocument:

- DataObjectPtr is an instance of Type PurchaseOrderType.
- RootElementURI is null because the XSD has no targetNamespace URI.
- RootElementName is purchaseOrder.
- Encoding is null because the document did not specify an encoding.
- XMLDeclaration is true because the document contained an XML declaration.
- XMLVersion is 1.0.
- SchemaLocation and noNamespaceSchemaLocation are null because they are not specified in the document.

XSDHelper

An XSDHelper provides additional information when a Type or Property is defined by an XML Schema (XSD). Also, an XSDHelper can define Types from XSDs.

An XSD Helper is accessed via a static method of the class HelperProvider, and is connected to the DataFactory by passing the factory as a creation parameter, thus:

```
XSDHelperPtr myXSDHelper = HelperProvider::getXSDHelper(myDataFactory);
```

If SDO Types and Properties were not originally defined by an XSD, or if the original XSD declaration information is not available, the helper methods will return null or false. IsXSD() can be used to tell if the XSDHelper has information about a Type.

The original name and namespace from an XML Schema can found using the getLocalName() and getNamespaceURI() methods.

It is possible to tell if a Property is serialized as an XML element or attribute with the isElement() and isAttribute() methods.

XML Schema global elements and attributes can be found using the getGlobalProperty() method. This is the most common way to build XML documents with open content, by finding a global property with the XSDHelper and then setting the property on an open content DataObject.

XSD Appinfo may be returned for a DataObject through the getAppinfo() methods. Appinfo is commonly used to specify information specific to a service in an XSD that may be valuable for configuring that service.

Generating XSDs

The XSDHelper can generate a new XSD for Types that do not already have an XSD definition. This is useful when the source of the Types come from services in another domain, such as relational databases, programming languages and UML. The generated XSD format is described later in the Generation of XSD section.

If an XML Schema was originally used to define the Types, that original XSD should be used instead of generating a new XSD. If a new XML Schema is generated when one already exists, the generated schema and the original schema will **not** be compatible and will validate different documents. The XMLHelper will follow the original XSD if one exists, otherwise it will follow a generated XSD.

XSDHelper API

---

const char defineFile(const char* schemaFile) ;*
const char define(std::istream& schema) ;*
const char define(const char* schema) ;*

Populates the data factory with Types and Properties from the schema
Loads from file, stream or char* buffer.
The return value is the URI of the root Type

---

char generate(*
const TypeList& types,
const char targetNamespaceURI = "") ;*

void generate(
const TypeList& types,
std::ostream& outXsd,
const char targetNamespaceURI = "") ;*

---

void generateFile(
const TypeList& types,
const char fileName,*
const char targetNamespaceURI = "") ;*

DataFactoryPtr getDataFactory() ;

```
const char* getLocalName(const Type& type);
```

```
const char* getLocalName(const Property& property);
```

```
const char* getNamespaceURI(const Property& property);
```

```
bool isAttribute(const Property& property);
```

```
bool isElement(const Property& property);
```

```
bool isMixed(const Type& type);
```

```
bool isXSD(const Type& type);
```

```
const Property& getGlobalProperty(const char* uri, const char* propertyName,
bool isElement);
```

```
const char* getAppinfo(const Type& type, const char* source);
```

```
const char* getAppinfo(const Property& property, const char* source);
```

Exceptions

All exceptions thrown by the SDO code are derived from the base exception SDORuntimeException. The list of subclasses is:

SDOOutOfMemoryException – An allocation failed.
SDOPathNotFoundException - Path to a data object was not valid.
SDOPropertyNotFoundException – The property requested from a Type is not valid.
SDOTypeNotFoundException- A Type was not available in the factory.
SDOUnsupportedOperationException – Actions such as getList on a single-value.
SDOInvalidConversionException – Actions such as getInteger on a containment DataObject.
SDOXMLParserException - an error occured parsing the XML input

Exception API

---

const char getEClassName();*

Returns the type of the exception as a string. E.g "SDOInvalidConversionException".

---

const char getMessageText();*

Returns the message which the SDO runtime attached to the exception.

---

const char getFileName();*

Returns the filename within which the exception occurred – May be zero if not using the running the debug build.

---

unsigned long getLineNumber();

Returns the line number at which the exception occurred.

---

const char getFunctionName() const;*

Returns the function name in which the exception occurred.

C++ Type Safe Interface Specification

Data Objects (DOs) may be dynamic or static. When they are static DOs, an interface class can be generated following the pattern described in the tables below. The implementation of the static interfaces must also implement the `DataObject` interface, enabling all DOs, whether static or dynamic, to be used with the `DataObject` interface. The behavior of implementations of these interfaces must be identical whether called through the generated or DataObject interfaces.

Each Type generates one interface class. When [propertyName] and [typeName] appear, the first letter is capitalized. Each row in the code generation template below specifying a method is generated when the expression for the property in the left column is true.

The C++ namespace defaults to "noNamespace". Where the SDO source is an XSD the targetNameSpace will be used as the C++ namespace. This may require some transformation to generate a valid C++ namespace.

Generated Types may only be defined for Types where type.isDataType() is false.

C++ code generation when the SDO source comes from an XSD uses the sdo annotations to determine the mapping. Because the names used are the same as in the XSD, it is often important to annotate the XSD with sdo:name to produce valid C++ code. All SDO C++ generators using the same annotated XSD as input will produce the same C++ interfaces when measured by invocation compliance above.

Each Type has a constructor that takes a DataObject* as a parameter to cast the data object to the Type. An exception should be thrown where the specified data object is not of the correct type.

Code generation template

Type	C++
For each Property in type.getProperties():	`class [typeName]` `{`
many = false && [propertyType] != boolean	`[propertyType] get[propertyName]();`
many = false && [propertyType] = boolean	`bool is[propertyName]();`
many = false && readOnly = false	`void set[propertyName]([propertyType]);`
many = true	`SDOList<[propertyType]>` `get[propertyName]();`

where

- [typeName] = type.getName() with the first character Character.toUpperCase().

- [propertyName] = property.getName()with the first character Character.toUpperCase().
- [propertyType] = property.getType().getName()
- SDOList<[propertyType]> is a templated list for accessing many-valued properties

For convenience, code generators may at their discretion use the following pattern for a typed create method when the property's type is a DataObject type:

- `[Type] create[propertyName]()`

This method is identical in behavior to DataObject.create([propertyName]).

For convenience, code generators may at their discretion use the following pattern for isSet/unset methods:

- `bool isSet[propertyName]()`
- `void unset[propertyName]()`

These methods are identical in behavior to DataObject.isSet([propertyName]) and DataObject.unset([propertyName]).

These convenience options are not required to be offered by compliant SDO C++ code generators.

SDOList template class

For many-valued properties the *get[PropertyName]()* methods return a templated list class of the appropriate Type. This can be used in the same way as a DataObjectList.

```
template <typename Type>
class SDOList
{
      public:
      unsigned int size();
  Type operator[] (unsigned int index);
  void insert(unsigned int index, Type type);
  void append(Type type);
}; // end SDOList
```

Example

For the company XSD without any annotations, the following are the minimal C++ interfaces generated:

```
namespace companyNS
{
```

```
     class CompanyType
     {
     public:
     CompanyType(DataObject* dataObject);
   const char* getName();
   void setName(const char* value);
   SDOList<DepartmentType> getDepartments();
   EmployeeType getEmployeeOfTheMonth();

 }; // end CompanyType

class DepartmentType
{
     public:
     DepartmentType (DataObject* dataObject);
   const char* getName();
   void setName(const char* value);
   const char* getLocation();
   void setLocation(const char* value);
   int getNumber();
   void setNumber(int value);
   SDOList<EmployeeType> getEmployees();
     }; // end DepartmentType
```

```
      class EmployeeType
  {
      public:
      EmployeeType (DataObject* dataObject);
     const char* getName();
     void setName(const char* value);
     const char* getSN();
     void setSN(const char* value);
     bool isManager();
     void setManager(bool value);
       }; // end DepartmentType


} // end namespace
```

Example usage

A simple code sample that prints the names of all the departments:

```
...
using namespace companyNS;
...
// Assuming we have obtained the root DataObject from a
// data access service
CompanyType myCo = (CompanyType)rootObject->getDataObject("company");
SDOList<DepartmentType> departments = myCo.getDepartments();
for (int i=0; i<departments.size(); i++)
{
   cout << "Department " << i << ": "
        << departments[i].getName() // invoke getName on DepartmentType
        << endl;
}
```

Standard SDO Types

These are the predefined SDO Types that are always available from either:

- TypeHelper.getType("commonj.sdo", const char* typeName).
- DataGraph.getType("commonj.sdo", const char* typeName).

SDO Data Types

The term *SDO data type* refers to an SDO Type where isDataType() = true. None of the types have any Properties unless noted. All values are false unless noted.

SDO Model Types

Type and Property describe themselves. The definition is:

SDO Model Types

```
Type
  name="Type"
  open=true
  uri="commonj.sdo"

  Property name="baseType" many=true type="Type"
  Property name="property" containment=true many=true
  Property name="aliasName" many=true type="String"
  Property name="name" type="String"
  Property name="uri" type="String"
  Property name="dataType" type="Boolean"
  Property name="open" type="Boolean"
  Property name="sequenced" type="Boolean"
  Property name="abstract" type="Boolean"
type="Property"
```

```
Property
  name="Property"
  open=true
  uri="commonj.sdo"

  Property name="aliasName" many=true type="String"
  Property name="name" type="String"
  Property name="many" type="Boolean"
  Property name="containment" type="Boolean"
  Property name="type" type="Type"
  Property name="default" type="Object"
  Property name="readOnly" type="Boolean"
  Property name="opposite" type="Property"
```

SDO Type and Property constraints

There are several restrictions on SDO Types and Properties. These restrictions ensure Types and Properties for DataObjects are consistent with their API behavior. Behavior of ChangeSummaryType Properties is defined.

- Instances of Types with dataType=false must implement the DataObject interface and have isInstance(DataObject) return true.
- Values of bidirectional Properties with type.dataType=false and many=true must be unique objects within the same list.
- Values of Properties with type.dataType=false and many=true cannot contain null.
- Property.containment has no effect unless type.dataType=false
- Property.default!=null requires type.dataType=true and many=false
- Types with dataType=true cannot contain properties, and must have open and sequenced=false.
- Type.dataType and sequenced must have the same value as their base Types' dataType and sequenced.
- Type.open may only be false when the base Types' open are also false.
- Instance classes in Java must mirror the extension relationship of the base Types.
- Properties that are bi-directional require type.dataType=false
- Properties that are bi-directional require that no more than one end has containment=true
- Properties that are bi-directional require that both ends have the same value for readOnly
- Properties that are bi-directional with containment require that the non-containment Property has many=false.
- Names and aliasNames must all be unique within Type.getProperties()

ChangeSummaryType Properties:

- Types may contain one property with type ChangeSummaryType.

- A property with type ChangeSummaryType must have many=false and readOnly=true.
- The scope of ChangeSummaries may never overlap. The scope of a ChangeSummary for a DataGraph is all the DataObjects in the DataGraph. If a DataObject has a property of type ChangeSummary, the scope of its ChangeSummary is that DataObject and all contained DataObjects. If a DataObject has a property of type ChangeSummay, it cannot contain any other DataObjects that have a property of type ChangeSummay and it cannot be within a DataGraph.
- ChangeSummaries collect changes for only the DataObjects within their scope.
- The scope is the same whether logging is on or off.
- Serialization of a DataObjects with a property of type ChangeSummaryType follows the normal rules for serializing a ChangeSummary.

Example Use of ChangeSummaryType

A common use of defining DataObject Types with a ChangeSummary is when wrapping specific existing types such as PurchaseOrders along with a ChangeSummary tracking their changes. A message header defined by the following XSD is an example.

```
<element name="message" type="PurchaseOrderMessageType"/>
<complexType name="PurchaseOrderMessageType">
  <sequence>
    <element name="purchaseOrder" type="po:PurchaseOrderType"/>
    <element name="changeSummary" type="sdo:ChangeSummaryType"/>
  </sequence>
</complexType>
```

The following is an example message document:

```
<message>
  <purchaseOrder orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
    </shipTo>
    <comment>Hurry, my lawn is going wild!</comment>
  </purchaseOrder>
  <changeSummary>
    <USAddress sdo:ref="sdo:/purchaseOrder.0/shipTo" name="John Public"/>
  </changeSummary>
</message>
```

XML Schema to SDO Mapping

XML Schema declarations (XSD) are mapped to SDO Types and Properties following the principles outlined below. [2] [7] (The abbreviation XSD is used for both the XML Schema infoset and the XML Schema declarations used to build the infoset.)

This simple yet flexible mapping allows SDO DataObjects to represent XML documents following an XSD. The vast majority of XSD capabilities are mapped and several corner cases are included. XML documents without XSDs are also handled.

Sequenced DataObjects preserve detailed information about the order of XML elements.

This document describes the Mapping Principles, Mapping of XSD Types, Sequenced DataObject, Mapping of XSD elements and Attributes, Mapping of data types and XML document mapping. It also provides Examples.

Mapping Principles

Creating SDO Types and Properties from XML Schema is increasingly important as a great deal of structured information is described by XSDs. The following tables describe the mapping.

XML Schema Concept	SDO Concept
Schema	URI for Types
Simple Type	Type, dataType=true SDO data types
Complex Type	Type, dataType=false SDO DataObjects
Attribute	Property within enclosing Type
Element	Property within enclosing Type

The general principles are that:

1. A Schema target namespace describes the URI for a set of Types.
2. SimpleType declarations describe data types, Types where isDataType() is true.
3. ComplexType declarations describe DataObjects, Types where isDataType() is false.
4. Within each ComplexType, the elements and attributes describe Properties in the corresponding enclosing Type.
5. Model groups (all, choice, sequence, group reference) are expanded in place and do not describe Types or Properties. There is no SDO construct corresponding to groups in this specification.
6. Open content and mixed content map to Type.open.
7. Mixed content maps to Type.sequenced and uses the Text property for mixed text.
8. Order of element content maps to Type.sequenced.

9. XSD any and anyAttribute (wildcard) declarations are not required to map to Types or Properties.
10. We do not allow design changes that complicate the simple cases to solve more advanced cases.
11. The mapping input is an annotated XSD using the SDO annotations. The mapping output is SDO Types and Properties.
12. Normally, SDO names are the same as the XSD names. To change the SDO name user can annotate an XSD with sdo:name annotations. In some cases, for example in the case of duplicate component names in XSD, the original XSD names cannot be preserved in SDO. In such cases, an SDO-aware code generator tool will generate new names and virtually add sdo:name annotations to the original XSD. Then, the tool will use the Annotated Schema to generate SDO. Such tool must be able to serialize the Annotated Schema at user request.
13. This mapping specifies a minimum. Implementations may expand this mapping to perform additional functions as long as the mapping stated here works for all client code.

Mapping of XSD to SDO Types and Properties

There are a number of customizations that can be used to improve the mapping to SDO.

These are expressed as attributes in the SDO namespace for XML, "commonj.sdo/xml". The following XSD attributes in the SDO XML namespace are used to modify the constructed SDO model:

1. **name** - sets the SDO name to the name specified here. Applies to Type and Property. Used in ComplexType, SimpleType, element, and attribute declarations. The XSD type of the annotation is string.
2. **propertyType** - sets the Property's Type as specified by the QName value. Applies to Property. Used in element and attribute declarations. Property.type.dataType must be false. The XSD type must be IDREF, IDREFS, or anyURI, or restrictions of these types. The XSD type of the annotation is QName.
3. **oppositeProperty** - sets the Property opposite to be the property with the given name within the Type specified by **propertyType**. Applies to Property, making the property bidirectional. Used in element and attribute declarations. Property.type.dataType must be false. The XSD type must be IDREF, IDREFS, or anyURI or restrictions of these types. Requires sdo:propertyType on the property. Automatically creates the opposite property if one or both ends are specified in the XSD, with opposites bidirectional. The XSD type of the annotation is string.
4. **sequence**="true" - sets Type.sequenced to true. Applies to Type. Used in ComplexType declarations. A Sequenced Type has a Sequence for all XML Elements. The default is false. If schema extension is used, the base complexType must also be marked sequence="true". The XSD type of the annotation is boolean.

5. **string="true" -** sets the SDO Type to String for XSD SimpleTypes as a means to override the instance class when the exact values must be preserved. Applies to Property. Used in element and attribute declarations. Same as sdo:dataType="sdo:String". The XSD type of the annotation is boolean.
6. **dataType** - sets the Property's type as specified by the QName value as a means to override the declared type. Applies to XML attributes and elements with simple content. Used in element and attribute declarations. The XSD type of the annotation is QName.
7. **aliasName** - add alias names to the SDO Type or Property. The format is a list of names separated by whitespace, each becoming an aliasName. Applies to Type and Property. The XSD type of the annotation is string.
8. **readOnly** - indicate the value of Property.readOnly. The format is boolean with default false. Applies to Property. Used in element and attribute declarations. The XSD type of the annotation is boolean.

In all tables, SDO Type and Property values that are not shown default to false or null, as appropriate. [URI] is the targetNamespace. Use sdo:name to override the names as desired.

XML Schemas

XML Schemas	SDO Package
Schema with targetNamespace `<schema targetNamespace=[URI]>`	[URI] is type.uri) for the types defined by this Schema.
Schema without targetNamespace `<schema>`	[URI] is null. Null is type.uri for the types defined by this Schema.

XML Simple Types

XML simple types map directly to SDO types.

The mapping of XML Schema built-in simple types is defined in another section below. The Java instance class is the class for the values returned by DataObject.get(property). The notation [BASE].instanceClass indicates the instance class of the SDO Type corresponding to [BASE]. When deriving Simple Types by restriction, the base for the SDO Type follows the XSD SimpleType restriction base.

When the XSD type is integer, positveInteger, negativeInteger, nonPositiveInteger, nonNegativeInteger, long, or unsignedLong, and there are facets (minInclusive, maxInclusive, minExclusive, maxExclusive, enumeration) constraining the range to be within the range of int, then the Java instance class is int and the base is null unless the base Type's instance class is also int.

XML Simple Types	SDO Type	Comments

Simple Type with name
 	
`<simpleType name=[NAME]>`
`  <restriction base=[BASE]/>`
`</simpleType>`
	`Type name=[NAME]`
	`  base=[BASE]`
	`  dataType=true`
	`  uri=[URI]`	

Simple Type Anonymous
 	
`<... name=[NAME] ...>`
`<simpleType>`
`  <restriction base=[BASE]/>`
`</simpleType>`
`</...>`
 	
[NAME]=enclosing element or attribute name
	`Type name=[NAME]`
	`  base=[BASE]`
	`  dataType=true`
	`  uri=[URI]`
	
	• [NAME] of the anonymous type is the same as the name of the enclosing element or attribute declaration.	

Simple Type with sdo:name
 	
`<simpleType name=[NAME]`
`  sdo:name=[SDO_NAME]>`
`  <restriction base=[BASE]/>`
`</simpleType>`
	`Type name=[SDO_NAME]`
	`  base=[BASE]`
	`  dataType=true`
	`  uri=[URI]`	

`Simple Type with abstract`
 	
`<simpleType name=[NAME]`
`  abstract="true">`
`  <restriction base=[BASE]/>`
`</simpleType>`
	`Type`
	`  abstract=true`
	`  base=[BASE]`
	`  dataType=true`
	`  uri=[URI]`	

Simple Type with list of itemTypes
 	
`<simpleType name=[NAME]>`
`  <list itemType=[BASE] />`
`</simpleType>`
	`Type name=[NAME]`
	`  dataType=true`
	`  uri=[URI]`	

Simple Type with union
 	
`<simpleType name=[NAME]>`
`  <union memberTypes=[TYPES]/>`
`</simpleType>`
	`Type name=[NAME]`
	`  dataType=true`
	`  uri=[URI]`	

XML Complex Types

XML Complex Types	SDO Type	

Complex Type with empty content
 	
`<complexType name=[NAME] />`
	`Type name=[NAME]`
	`  uri=[URI]`
	
	No Properties.	

Complex Type with content
 	
`<complexType name=[NAME] />`
	`Type name=[NAME]`
	`  uri=[URI]`
	
	Properties for each element and attribute.	

Complex Type Anonymous
 	
`<... name=[NAME] ...>`
`  <complexType  />`
`</...>`
 	
[NAME]=enclosing element name
	`Type name=[NAME]`
	`  uri=[URI]`
	
	• [NAME] of the anonymous type is the same as the name of the enclosing element declaration	

Complex Type with sdo:name
 	
`<complexType name=[NAME]`
`  sdo:name=[SDO_NAME] />`
	`Type name=[SDO_NAME]`
	`  uri=[URI]`	

Complex Type with abstract
 	
`<complexType name=[NAME}`
`  abstract="true">`
	`Type name=[NAME]`
	`  abstract=true`
	`  uri=[URI]`	

Complex Type with sdo:aliasName
 	
`<complexType name=[NAME]`
 	
`sdo:aliasName=[ALIAS_NAME]`
`/>`
	`Type name=[NAME]`
	`  aliasName=[ALIAS_NAME]`
	`  uri=[URI]`	

Complex Type extending a Complex Type
 	
`<complexType name=[NAME]>`
`  <complexContent>`
`  <extension base=[BASE]/>`
`  </complexContent>`
`</complexType>`
 	
or
 	
`<complexType name=[NAME]>`
`  <simpleContent>`
`  <extension base=[BASE]/>`
`  </simpleContent>`
`</complexType>`
	`Type name=[NAME]`
	`  base=[BASE]`
	`  uri=[URI]`
	
	`properties+=[BASE].properties`
	
	• Type.getProperties() maintains the order of [BASE].getProperties() and appends the Properties defined here.	

Complex Type with complex content restricting a Complex Type
 	
`<complexType name=[NAME]>`
`  <complexContent>`
`  <restriction base=[BASE]/>`
`  </complexContent>`
`</complexType>`
	`Type name=[NAME]`
	
	`properties=[BASE].properties`
	`  base=[BASE]`
	`  uri=[URI]`
	
	• Type.getProperties() maintains the order of [BASE].getProperties() and appends the Properties defined here.
	• When element and attribute declarations are in both the base type and the restricted type, no additional Properties are created and declarations inside the complex type are ignored.
	• When new element or attribute declarations are added in the restricted type that are not in the base type and restrict wildcard <any> and <anyAttribute> in the base, the element and attribute declarations are added as new Properties.	

Complex Type with simple content restricting a Complex Type
 	
`<complexType name=[NAME]>`
`  <simpleContent>`
`  <restriction base=[BASE]/>`
`  </simpleContent>`
`</complexType>`
	`Type name=[NAME]`
	`  base=[BASE]`
	`  uri=[URI]`
	
	`properties+=[BASE].properties`
	
	• Type.getProperties() maintains the order of [BASE].getProperties() and appends the Properties defined here.	

Complex Type with mixed content
 	
`<complexType name=[NAME]`
`  mixed="true" />`
	`Type name=[NAME]`
	`  open=true`
	`  sequenced=true`
	`  uri=[URI]`
	
	• The SDO Text Property is used for mixed text as an instance property.
	• Use getInstanceProperties() for reflection.
	• Sequence is typically used to access the values. DataObject and generated accessors also may be used.	

Complex Type with sdo:sequence
 	
`<complexType name=[NAME]`
`  sdo:sequence="true" />`
	`Type name=[NAME]`
	`  sequenced=true`
	`  uri=[URI]`	

Complex Type extending a SimpleType
 	
`<complexType name=[NAME]>`
`  <simpleContent>`
`  <extension base=[BASE]/>`
`  </simpleContent>`
`</complexType>`
	`Type name=[NAME]`
	`  uri=[URI]`
	`  Property:`
	`  name="value" type=[BASE]`
	
	• Properties are created for attribute declarations.	

Complex Type with open content
 	
`<complexType name=[NAME]>`
`  ...`
`  <any />`
`  ...`
`</complexType>`
	`Type name=[NAME]`
	`  open=true`
	`  uri=[URI]]`
	
	• No property required for <any>.
	• Use getInstanceProperties() for reflection.
	• DataObject and generated accessors also may be used to access the value.
	• If maxOccurs > 1, sequenced=true.	

Complex Type with open attributes
 	
`<complexType name=[NAME]>`
`  ...`
`  <anyAttribute />`
`  ...`
`</complexType>`
	`Type name=[NAME]`
	`  open=true`
	`  uri=[URI]`
	
	• No property required for <anyAttribute>.
	• Use getInstanceProperties() for reflection.
	• DataObject and generated accessors also may be used to access the value.	

Mapping of XSD Attributes and Elements to SDO Properties

Each XSD element or attribute maps to an SDO property.

The Property.containingType is the SDO Type for the enclosing ComplexType declaration.

The order of Properties in Type.getDeclaredProperties() is the lexical order of declarations in the XML Schema ComplexType. When extension is used, the Properties of the base type occur first in the Properties list.

If elements and attributes within a complexType, and its base types, have the same local name then unique names must be assigned by sdo:name. This ensures that all property names in Type.getProperties() are unique. Multiple elements with the same name and URI are combined into a single Property and the Type is sequenced, as described in the Mapping of XSD Elements section.

When creating a Property where the default or fixed value is not defined by the XSD, the Property's default is assigned based on its Type's instance class, property.getType().getInstanceClass() :

- Boolean has default false.
- Primitive numerics (Byte, Char, Double, Float, Int, Short, Long) have default is 0.
- Otherwise, the default is null.

Note that XSD anyType is a ComplexType and XSD anySimpleType is a SimpleType. They follow the normal mapping rules.

Mapping of XSD Attributes

XML Attribute	SDO Property

Attribute
 	
`<attribute name=[NAME]`
`  type=[TYPE] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	• DataObject accessors may enforce simple type constraints.

Attribute with sdo:name
 	
`<attribute name=[NAME]`
`  sdo:name=[SDO_NAME]`
`  type=[TYPE] />`
	`Property name=[SDO_NAME]`
	`  type=[TYPE]`

Attribute with sdo:aliasName
 	
`<attribute name=[NAME]`
`  sdo:aliasName=[ALIAS_NAME]`
`  type=[TYPE] />`
	`Property name=[NAME]`
	`  aliasName=[ALIAS_NAME]`
	`  type=[TYPE]`

Attribute with default value
 	
`<attribute name=[NAME]`
`  type=[TYPE]`
`  default=[DEFAULT] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	`  default=[DEFAULT]`

Attribute with fixed value
 	
`<attribute name=[NAME]`
`  type=[TYPE]`
`  fixed=[FIXED] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	`  default=[FIXED]`

Attribute reference
 	
`<attribute ref=[ATTRIBUTE] />`
	`Property name=[ATTRIBUTE].[NAME]`
	`  type=[ATTRIBUTE].[TYPE]`
	`  default=[ATTRIBUTE].[DEFAULT]`
	• Use the XSDHelper to determine the URI of the attribute if the referenced attribute is in another namespace.

Attribute with sdo:string
 	
`<attribute name=[NAME]`
`  type=[TYPE]`
`  sdo:string="true" />`
	`Property name=[NAME]`
	`  type=String`
	• The type of the property is SDO String
	• Used when the instance class for TYPE is not appropriate.

Attribute referencing a DataObject with sdo:propertyType
 	
`<attribute name=[NAME]`
`type=[TYPE]`
`sdo:propertyType=[P_TYPE] />`
 	
where `[TYPE] = IDREF, IDREFS, anyURI` or restrictions of these types.
	`Property name=[NAME]`
	`type=[P_TYPE]`
	`many=true` (for IDREFS only)

Attribute with bi-directional property to a DataObject with sdo:oppositeProperty and sdo:propertyType
 	
`<attribute name=[NAME]`
`type=[TYPE]`
`sdo:propertyType=[P_TYPE]`
 	
`sdo:oppositeProperty=[PROPERTY] />`
 	
where:
`[TYPE] = IDREF, IDREFS, anyURI` or restrictions of these types.
	`Property name=[NAME]`
	`type=[P_TYPE]`
	`opposite=[PROPERTY]`
	`many=true` (for IDREFS only)
	
	Declared on:
	`Type [P_TYPE]:`
	`Property name=[PROPERTY]`
	`type=[NAME].containingType`
	`opposite=[NAME]`
	`containingType=[P_TYPE]`

Attribute with sdo:dataType
 	
`<attribute name=[NAME]`
`type=[TYPE]`
`sdo:dataType=[SDO_TYPE] />`
	`Property name=[NAME]`
	`type=[SDO_TYPE]`
	• The type of the property is the SDO type for [SDO_TYPE]
	• Used when the instance class for TYPE is not appropriate.

XML Global Elements and Attributes	SDO Property
Global Element `<element name=[NAME] />`	Same as local element declaration except the containing Type is not specified by SDO other than the Type's URI is the XSD target namespace.
Global Attribute `<attribute name=[NAME] />`	Same as local attribute declaration except the containing Type is not specified by SDO other than the Type's URI is the XSD target namespace.

Mapping of XSD Elements

If a ComplexType has content with two elements that have the same local name and the same targetNamespace, whether through declaration, extension, substitution, groups, or other means, the duplication is handled as follows:

- The ComplexType becomes a sequenced type, as if sdo:sequence="true" was declared.
- A single property is used for all the elements with the same local name and the same targetNamespace, where isMany=true.
- The type of the property is SDO Object.
- When substitution is possible for a Type, Type.open is true.

If schema extension is used, the base type may need to be modified with sdo:sequence="true" and elements with name conflicts introduced in extensions require that the property in the extended base type must be made isMany=true.

XML Elements	SDO Property
Element `<element name=[NAME] />`	`Property name=[NAME]`

Element with sdo:name
 	
`<element name=[NAME]`
`  sdo:name=[SDO_NAME] />`
	`Property name=[SDO_NAME]`

Element with sdo:aliasName
 	
`<element name=[NAME]`
`  sdo:aliasName=[ALIAS_NAME]`
`  type=[TYPE] />`
	`Property name=[NAME]`
	`  aliasName=[ALIAS_NAME]`
	`  type=[TYPE]`

Element reference
 	
`<element ref=[ELEMENT] />`
	`Property name=[ELEMENT].[NAME]`
	`type=[ELEMENT].[TYPE]`
	`default=[ELEMENT].[DEFAULT]`
	• Use the XSDHelper to determine the URI of the element if the referenced element is in another namespace.

Element with maxOccurs > 1
 	
`<element name=[NAME]`
`maxOccurs=[MAX] />`
 	
where `[MAX] > 1`
	`Property name=[NAME]`
	`many=true`

Element in all, choice, or sequence
 	
`<[GROUP] maxOccurs=[G_MAX]>`
`<element name=[NAME]`
`type=[TYPE]`
`maxOccurs=[E_MAX] />`
`</[GROUP] >`
 	
where
[GROUP] = all, choice, sequence
• Element groups and model groups are treated as if they were expanded in place.
• Nested [GROUP]s are expanded.
	`Property name=[NAME]`
	`type=[TYPE]`
	`many=true`
	
	Type sequenced=true
	
	• A property is created for every element
	• many=true when E_MAX or G_MAX is > 1
	• sequenced=true if the content allows elements to be interleaved. (for example <A/><B/><A/>)
	• sequenced=true if G_MAX > 1 and there is more than one element in this group or a contained group.
	• Property declarations are the same whether group is <all> or <choice> or <sequence>
	• Property behavior ignores group declarations.
	• Validation of DataObjects for the group constraints is external to the DataObject interface.

Element with nillable
 	
`<element name=[NAME]`
`nillable="true" type=[TYPE]/>`
	`Property name=[NAME]`
	
	• If the type of the element has Simple Content without attributes, a Java Type with an Object instance class is assigned. For example, IntObject instead of Int.
	• In an XML document, xsi:nil="true" corresponds to a null value for this property.

Element with substitution group
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  substitutionGroup=[ELEMENT] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	
	• Use getInstanceProperties() for reflection.
	• Use get(property) to access the value.

Elements of Complex Type follow this table, in addition.

XML Elements with Complex Type	SDO Property

`<element name=[NAME]`
`  type=[TYPE] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	`  containment=true`

Elements of Simple Type follow this table, in addition.

XML Elements with Simple Type	SDO Property

Element of SimpleType
 	
`<element name=[NAME]`
`  type=[TYPE] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	
	• DataObject accessors may enforce simple type constraints.

Element of SimpleType with default
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  default=[DEFAULT] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	`  default=[DEFAULT]`

Element of SimpleType with fixed
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  fixed=[FIXED] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	`  default=[FIXED]`

Element of SimpleType with sdo:string
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  sdo:string="true" />`
	`Property name=[NAME]`
	`  type=String`
	• The type of the property is SDO String
	• Used when the instance class for TYPE is not appropriate.

Element referencing a DataObject with sdo:propertyType
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  sdo:propertyType=[P_TYPE] />`
 	
where `[TYPE] = IDREF, IDREFS, anyURI` or restrictions of these types
	`Property name=[NAME]`
	`  type=[P_TYPE]`
	`  many=true` (for IDREFS only)

Element with bi-directional reference to a DataObject with sdo:propertyType and sdo:oppositeProperty
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  sdo:propertyType=[P_TYPE]`
 	
`sdo:oppositeProperty=[PROPERTY]`
`/>`
 	
where `[TYPE] = IDREF, IDREFS, anyURI` or restrictions of these types
	`Property name=[NAME]`
	`  opposite=[PROPERTY]`
	`  type=[P_TYPE]`
	`  many=true` (for IDREFS only)
	
	Declared on `Type PR_TYPE]:`
	`Property name=[PROPERTY]`
	`  type=[NAME].containingType`
	`  opposite=[NAME]`
	`  containingType=[P_TYPE]`

Element of SimpleType with sdo:dataType
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  sdo:dataType=[SDO_TYPE] />`
	`Property name=[NAME]`
	`  type=[SDO_TYPE]`
	• The type of the property is the SDO type for [SDO_TYPE]
	• Used when the instance class for TYPE is not appropriate.

XML Schema Element special types	**SDO Property**

Element with type SDO ChangeSummaryType
 	
`<element name=[NAME]`
`  type="sdo:ChangeSummaryType"/>`
	`Property name=[NAME]`
	`  type=ChangeSummaryType`
	`  readOnly=true`

Conversion between XSD QName and SDO URI

When an XML document is loaded, a value of type xsd:QName is converted into an SDO URI with a value of:

- The namespace name + # + local part

where + indicates string concatenation.

When an XML document is saved, a value of type SDO can be converted back to an xsd:Qname, if that is the expected XML type:

- The URI value is parsed into two parts:
  - The namespace name is the URI up to but not including the last # character in the URI value.
  - The local part is the URI after the last # character in the URI value.
- An XML namespace declaration for a namespace prefix is made in the XML document. The declaration may be made at any enclosing point in the document in an implementation-dependent manner or an existing declaration may be reused.
- The declaration is of the form xmlns:prefix="namespace name".
- The prefix is implementation-dependent.
- The QName value is of the form prefix:local part.

Example:

- Message is a property of XSD type QName and SDO type URI
- Load: <input message="tns:inputRequest" name="inputMessage" xmlns:tns="http://example.com" />
- inputDataObject.get(message) returns http://example.com#inputRequest
- inputDataObject.set(message, "http://test.org#testMessage")
- Save: <input message="tns:testMessage" name="inputMessage" xmlns:tns="http://test.org" />

Examples of XSD to SDO Mapping

XSD	SDO

Schema declaration
 	
`<schema targetNamespace= "http://`
`www.example.com/IPO">`
	`uri="http://www.example.com/IPO"`

Global Element with Complex Type
 	
`<element name="purchaseOrder"`
`type="PurchaseOrderType"/>`
	`Property name="purchaseOrder"`
	`type="PurchaseOrderType"`
	`containment=true`

Global Element with Simple Type
 	
`<element name="comment"`
`type="xsd:string"/>`
	`Property name="comment" type="sdo:String"`

Complex Type
 	
`<complexType`
`name="PurchaseOrderType">`
	`Type name="PurchaseOrderType" uri="http:/`
	`/www.example.com/IPO"`

Simple Type
 	
`<simpleType sdo:name="QuantityType">`
`<restriction`
`base="positiveInteger">`
`<maxExclusive value="100"/>`
`</restriction>`
`</simpleType>`
 	
`<simpleType name="SKU">`
`<restriction base="string">`
`<pattern value="\d{3}-[A-Z]{2}"/>`
`</restriction>`
`</simpleType>`
	`Type name="QuantityType"`
	`dataType=true`
	`base="sdo:Int" uri="http://`
	`www.example.com/IPO"`
	
	`Type name="SKU" instanceClass="String"`
	`dataType=true uri="http://`
	`www.example.com/IPO"`
	`base="sdo:String"`

Local Element with Complex Type
 	
`<element name="shipTo"`
`type="ipo:Address"/>`
 	
`<element name="billTo"`
`type="ipo:Address"/>`
 	
`<element name="items"`
`type="ipo:Items"/>`
	`Property name="shipTo" type="Address"`
	`containment=true`
	`containingType="PurchaseOrderType"`
	
	`Property name="billTo" type="Address"`
	`containment=true`
	`containingType="PurchaseOrderType"`
	
	`Property name="items" type="Items"`
	`containment=true`
	`containingType="PurchaseOrderType"`

Local Element with Simple Type
 	
`<element ref="ipo:comment"`
`minOccurs="0"/>`
 	
`<element name="productName"`
`type="string"/>`
	`Property name="comment" type="String"`
	`containingType="PurchaseOrderType"`
	
	`Property name="productName" type="String"`
	`containingType="Items"`

Local Attribute
 	
`<attribute name="orderDate"`
`type="date"/>`
 	
`<attribute name="partNum"`
`type="ipo:SKU" use="required"/`
`>`
	`Property name="orderDate" type="Date"`
	`containingType="PurchaseOrderType"`
	
	`Property name="partNum" type="SKU"`
	`containingType="ItemType"`

Type extension
 	
`<complexType name="USAddress">`
`<complexContent>`
`<extension base="ipo:Address">`
	`Type name="USAddress" uri="http://`
	`www.example.com/IPO"`
	`base="ipo:Address"`

Local Attribute fixed value declaration
 	
`<attribute name="country"`
`type="NMTOKEN" fixed="US"/>`
	`Property name="country" type="String"`
	`default="US"`
	`containingType="USAddress"`

Multi-valued local element declaration
 	
`<element name="item" minOccurs="0"`
`maxOccurs="unbounded">`
`<complexType sdo:name="ItemType"/>`
`</element>`
	`Property name="item" type="ItemType"`
	`containment=true many=true`
	`containingType="Items"`
	
	`Type name="ItemType" uri="http://`
	`www.example.com/IPO"`

Attribute reference declarations
 	
`<attribute name="customer"`
`type="IDREF"`
`sdo:propertyType="cust:Customer"`
`sdo:oppositeProperty="purchaseOrder" />`
 	
`<attribute name="customer"`
`type="anyURI"`
`sdo:propertyType="cust:Customer"/>`
 	
`<attribute ref="xlink:href"`
`sdo:propertyType="cust:Customer" sdo:name="customer" />`
	`Property name="customer" type="Customer"`
	`opposite="Type[name='Customer']/`
	`property[name='purchaseOrder']"`
	`containingType="PurchaseOrderType"`
	
	Declared in the Customer type:
	`Property name="purchaseOrder"`
	`type="PurchaseOrderType"`
	
	`opposite="Type[name='PurchaseOrderType']/ property[name='customer']"`
	`containingType="Customer"`
	
	`Property name="customer" type="Customer"`
	`containingType="PurchaseOrderType"`
	
	`Property name="customer" type="Customer"`
	`containingType="PurchaseOrderType"`

Local Attribute ID declaration
 	
`<attribute name="primaryKey"`
`type="ID"/>`
	`Property name="primaryKey" type="String"`
	`containingType="Customer"`

Local Attribute default value declaration
 	
`<xsd:attribute name="country"`
`type="xsd:NMTOKEN"`
`default="US"/>`
	`Property name="country" type="String"`
	`default="US"`
	`containingType="USAddress"`

Abstract ComplexTypes
 	
`<complexType name="Vehicle"`
`abstract="true"/>`
	`Type name="Vehicle"`
	`abstract=true uri="http://`
	`www.example.com/IPO"`

SimpeType unions
 	
`<xsd:simpleType name="zipUnion">`
`<xsd:union`
`memberTypes="USState`
`listOfMyIntType"/>`
`</xsd:simpleType>`
	Type SDO Object is used as the Type for every Property resulting from elements and attributes with SimpleType zipUnion.

Notes:

1. Examples are from, or based on, IPO.xsd in http://www.w3.org/TR/xmlschema-0/
2. Type[name='Customer']/property[name='purchaseOrder'] refers to the declaration of the purchaseOrder Property in the Type Customer in the same document.

Example of SDO annotations

This example shows the use of sdo:stringand sdo:dataType.

```
<schema targetNamespace="http://www.example.com/IPO"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:ipo="http://www.example.com/IPO"

        xmlns:sdo="commonj.sdo/XML"
        >


 <complexType name="PurchaseOrderType" >
  <sequence>
   <element name="shipTo"     type="ipo:Address"/>
   <element name="billTo"     type="ipo:Address"/>
   <element ref="ipo:comment" minOccurs="0"/>
   <element name="items"      type="ipo:Items"/>
  </sequence>
  <attribute name="orderDate" type="date" sdo:dataType="ipo:MyGregorianDate"/>
 </complexType>

 <complexType name="Items">
  <sequence>
   <element name="item" minOccurs="0" maxOccurs="unbounded">
    <complexType sdo:name="Item">
     <sequence>
      <element name="productName" type="string"/>
      <element name="quantity" sdo:dataType="sdo:Int">
       <simpleType>
        <restriction base="positiveInteger">
         <maxExclusive value="100"/>
        </restriction>
       </simpleType>
      </element>
      <element name="USPrice"    type="decimal"/>
      <element ref="ipo:comment" minOccurs="0" sdo:aliasName="itemComment"/>
      <element name="shipDate"   type="date" minOccurs="0" sdo:string="true"/>
     </sequence>
     <attribute name="partNum" type="ipo:SKU" use="required"/>
    </complexType>
```

```
   </element>
  </sequence>
 </complexType>

 <simpleType name="MyGregorianDate" />

 <simpleType name="SKU" >
  <restriction base="string">
   <pattern value="\d{3}-[A-Z]{2}"/>
  </restriction>
 </simpleType>

</schema>
```

XML use of Sequenced Data Objects

Sequenced Data Objects are DataObjects with a sequence capturing the additional XML order information that is specific to XML documents.

Sequenced DataObjects have Type.sequenced=true. The XSD to SDO mapping defines an XML DataObject to be used when sdo:sequence="true" is declared in the XSD type.

The XML use of Sequenced DataObject defines a Sequence returned from the DataObject interface:

- getSequence() - A Sequence of all the elements and mixed text in the content of an XML element. Each entry in the Sequence represents either one XML element designated by the entry's Property, or XML mixed text, designated by the SDO Text Property. The name of the property is the same as the name of the XML element unless sdo:name was used to replace the name. The values of the entries are available through both the Sequence API and the DataObject API for the Properties. DataObject.getInstanceProperties() includes all the Properties in the Sequence. For open content, where XML any declarations, substitution groups or mixed content were used, the Properties of the entries might be declared in other Types than the DataObject's Type. The order of the entries in the Sequence is the same as the order of XML elements.

XSD Mapping Details

The following guidelines apply when mapping XSD to SDO:

1. The order of the Properties are declared within a Type is the lexical order of their declaration in an XSD. All Properties of the Type extended precede local declarations within the Type.
2. The XSD names are preserved in the Type and Property. Use the sdo:name override to modify names as an option to remove duplicate names, blank names, or names with special characters.
3. All declarations not covered in this Mapping may be ignored by a compliant implementation.
4. All <group> references, <attributeGroup> references, <include>s, and <import>s are fully expanded to the equivalent XSD as if these declarations were not present.
5. <choice> declarations for Complex Content are treated as <sequence> for the purpose of declaring Properties.
6. All comments, processing instructions, and annotations other than appinfo are discarded to the equivalent XSD as if these declarations were not present.
7. Redefinitions are expanded to the equivalent XSD as if these declarations were not present.
8. Model Groups (sequence, all, choice, group) do not contribute to the mapping except for maxOccurs>1 results in Properties with many=true.
9. Global group and attribute group declarations that include type declarations follow the normal mapping rules for those type declarations. The same types are used in all places the groups are referenced.

Compliance

The mappings here are the base mappings. Vendors may extend the mappings provided that client programs developed against these mappings continue to run. An SDO program using this mapping, and the DataObject, should be portable across vendor-added mappings and implementations.

Importing the sdo:alias annotation for XSDs is optional.

Corner cases

This specification does not standardize the mapping for corner cases. We follow the principle that complexity is never added to the simple cases to handle these more advanced cases. Future versions of SDO may define mappings for these corner cases.

1. List of lists without unions.
2. <element nillable="true" maxOccurs="unbounded" type="USAddress"/> Multi-valued nillable Properties with DataObject Types.
3. key and keyref.

4. When an element of anyType is used with xsi:type specifying simple content, a wrapper DataObject must be used with a property named "value" and type of SDO Object that is set to the wrapped type. For example, <element name="e" type="anyType"> and a document <e xsi:type="xsd:int">5</e> results in a wrapper DataObject where the value property is set to the Integer with value 5.
5. In some cases it is not possible to maintain an SDO base relationship when one exists in schema. This can happen for example when complex types extend simple types .
6. Elements that occur more than once and have type IDREFS and have sdo:propertyType will not be able to distinguish between consecutive elements in an XML document and one element with all the values in a single element. If there are interleaving elements sequence must be true to distinguish the order between elements. XML Schema recommends against the use of elements with type IDREF or IDREFS.
7. Anonymous type declarations in global group declarations, which are not a recommended schema design practice.

XML without Schema to SDO Type and Property

When a document does not have a schema, the following table and principles define the Types and Properties.

1. Each URI defines a DocumentRoot Type that contains all Properties that occur in the same URI.
2. Instances of these DocumentRoot Types will accept any Property in DataObject.get(property) and set(property).
3. If two XML elements or attributes have the same URI and Name, they are mapped to the same Property.
4. The URI is determined by the xmlns declarations in the document.
5. Mixed text is mapped to the SDO text Property.
6. The values allowed for Properties for XML attributes may not be DataObject.

XML Document	SDO Type and Property

XML elements
 	
`<[URI]:[NAME]>`
`...`
`</[URI]:[NAME]>`
	`Type name="DocumentRoot" sequenced=true uri=[URI] open=true`
	
	contains:
	
	`Property name=[NAME] type=DataObject containment=true many=true containingType=DocumentRoot`

XML attributes
 	
`<element [URI]:[NAME]="value"/>`
	`Type name="DocumentRoot" sequenced=true uri=[URI] open=true`
	
	contains:
	
	`Property name=[NAME] type=string containingType=DocumentRoot`

Generation of XSD from SDO Type and Property

When SDO Types and Properties did not originate from an XSD definition, it is often useful to define the equivalent XML schema declarations.

When an XSD is generated from Type and Property it contains all the information defined in the SDO Model. An XSD generated from Type and Property will round trip back to the original Type and Property. However, if the XSD was not generated and is used to create the Type and Property, regenerating the XSD will not round trip to produce the original. This is because there is more information in an XSD than in Type and Property, primarily focused on defining the XML document syntax.

The mapping principles are summarized in this table. A URI defines a schema and a target namespace. An SDO Type defines an XSD complex type and a global element declaration. An SDO property defines either a local element or an attribute in a complex type.

SDO	XSD
URI	<schema targetNamespace>
Type	<complexType> <element> global // or <simpleType>
Property	<element> local // or <attribute>

Each XSD contains Types with the same URI. When referring to other ComplexTypes, the implementation is responsible for generating the appropriate import and include XSD declarations.

An XSD can only be generated when:

1. Multiple inheritance is not used.
   - That is, all Types have no more than 1 base in Types.getBaseTypes().
2. The names of the Types and Properties are valid XSD identifiers.

The following defines the minimal XML schema declarations. When opening XML elements are shown the appropriate ending XML element is produced by the implementation. An implementation may include additional declarations as long as documents that validate with the

generated schema also generate with the customized schema. In addition, an implementation is expected to generate all required namespace declarations, includes, and imports necessary to produce a valid XML schema.

If a namespace declaration shown in the generation templates is not used by the XSD, it may be suppressed. Namespace declarations may have prefix names chosen by the implementation (instead of xsd, sdo, and tns). The specific elements containing the namespace declarations are determined by the implementation.

The schemas generated are a subset of the XMI 2.0 and 2.1 specifications. It is permissible to generate the xmi:version attribute from the XMI specification to enable XMI conformant software to read the XSDs and valid XML documents.

- The Schema element itself is generated with a target namespace determined by the URI of the Types that will be defined in the schema. `[URI]` is defined by type.uri. If [URI] is null then the XSD is generated without a targetNamespace.

SDO	XSD Schema

	`<xsd:schema`
	`xmlns:xsd="http://www.w3.org/2001/XMLSchema"`
	`xmlns:sdo="commonj.sdo"`
	`">`

`[URI]`	`xmlns:tns=[URI]` `targetNamespace=[URI]`

For each Type that is a dataType, type.dataType==true, an XSD SimpleType is generated. The SimpleType is based on the following:

- [NAME] is type.name
- [ABSTRACT] is type.abstract.
- [ALIAS_NAME] is space separated values from type.aliasNames and is produced if there are alias names.
- [BASE.NAME] is the name of the base type, type.getBaseTypes().get(0).getName() if not null. When not null, the simple type extends the base type. tns: is the prefix for the URI of the base type, type.getBaseTypes().get(0).getURI(). If the base type is in another namespace the appropriate namespace and import declarations are produced by the implementation. If there are no base types, then the xsd type used is from the table "Mapping of SDO DataTypes to XSD Built in Data Types" based on the instance class.

SDO Type	XSD SimpleType
	<xsd:simpleType name=[NAME]>
[ABSTRACT]	abstract="true"
[ALIAS_NAME]	Sdo:aliasName=[ALIAS_NAME]
[BASE.NAME]	<xsd:restriction base=tns:[BASE.NAME]>

For each Type that is not a dataType, type.dataType==false, an XSD ComplexType and a global element is generated. The ComplexType is based on the following:

- [NAME] is type.name
- [ABSTRACT] is type.abstract.
- [ALIAS_NAME] is space separated values from type.aliasNames and is produced if there are alias names.
- [BASE.NAME] is the name of the base type, type.getBaseTypes().get(0).getName() and is produced if not null. When produced, the complex type extends the base type. tns: is the prefix for the URI of the base type, type.getBaseTypes().get(0).getURI(). If the base type is in another namespace the appropriate namespace and import declarations are produced by the implementation.
- [SEQUENCED] indicates if the type is sequenced, type.sequenced. If true, the complex type declaration is mixed and the content of the element is placed in a <choice>. If false, the complex type contents are placed in a <sequence>. If no local elements are generated, the <choice> or <sequence> is suppressed.
- [OPEN] indicates if the type accepts open content, type.open. An <any> is placed in the content and <anyAttribute> is placed after the content.

SDO Type	XSD ComplexType
	<xsd:complexType name=[NAME]>
[ABSTRACT]	abstract="true"
[ALIAS_NAME]	sdo:aliasName=[ALIAS_NAME]
[BASE.NAME]	<xsd:complexContent> <xsd:extension base=tns:[BASE.NAME]>
[SEQUENCED]	mixed="true" <xsd:choice maxOccurs="unbounded">
![SEQUENCED]	<xsd:sequence>
[OPEN]	<xsd:any maxOccurs="unbounded" processContents="lax"/> <xsd:anyAttribute processContents="lax"/>

The global element for the type:

- lowercase(TYPE.NAME) is the type name with the first letter converted to lower case as defined type java.lang.Character.toLowerCase(). If two global elements with the same

name and target namespace would be generated when the lowercase is applied, then the original type name is used unchanged.
- [TYPE.NAME] is the type name type.name.

SDO Type	XSD Global Element
	`<xsd:element name=[lowercase(TYPE.NAME)] type=tns:[TYPE.NAME] />`

For each property in type.getDeclaredProperties(), either an element or an attribute will be generated, declared within the content of the property's containing type property.getContainingType(). An element is generated if either property.many or property.containment is true, or if property.get(xmlElement) is present and set to true. xmlElement is a property from XMLInfo. If the property is bidirectional and the opposite property has containment=true, nothing is generated. Otherwise, an attribute is generated. Round-trip between SDO models and their generated XSDs will preserve the order of the properties when all elements are generated.
- [NAME] is property.name
- [ALIAS_NAME] is space separated values from property.aliasNames and is produced if there are alias names.
- [READ_ONLY] is the value of property.readOnly and is produced if true.
- [MANY] indicates if property.many is true and maxOccurs is unbounded if true.
- [CONTAINMENT] indicates if property.containment is true.
    - When containment is true, then DataObjects of that Type will appear as nested elements in an XML document.
    - When containment is false and the property's type is a DataObject, a URI reference to the element containing the DataObject is used and an sdo:propertyType declaration records the target type. Values in XML documents will be of the form "#xpath" where the xpath is an SDO DataObject XPath subset. It is typical to customize the declaration to IDREF if the target element has an attribute with type customized to ID.
    - [TYPE.NAME] is the type of the element. If property.type.dataType is true, [TYPE.NAME] is the name of the XSD built in SimpleType corresponding to property.type, where the prefix is for the xsd namespace. Otherwise, [TYPE.NAME] is property.type.name where the tns: prefix is determined by the namespace declaration for the Type's URI.
- [OPPOSITE.NAME] is the opposite property if the property is bidirectional and indicated when property.opposite is not null.

SDO Property	XSD Element
	`<xsd:element name=[NAME] minOccurs="0"`

[ALIAS_NAME]	Sdo:aliasName=[ALIAS_NAME]
[READ_ONLY]	sdo:readOnly=[READ_ONLY]
[MANY]	maxOccurs="unbounded"
[CONTAINMENT]	type="tns:[TYPE.NAME]"
![CONTAINMENT]	type="xsd:anyURI" sdo:propertyType="tns:[TYPE.NAME]"
[OPPOSITE.NAME]	sdo:oppositeProperty=[OPPOSITE.NAME]

For all the properties in type.getDeclaredProperties() where the element test rules above indicate that an attribute is generated, a local attribute declaration is produced.

- [NAME] is property.name
- [ALIAS_NAME] is space separated values from property.aliasNames and is produced if there are alias names.
- [READ_ONLY] is the value of property.readOnly and is produced if true.
- [DEFAULT] is property.default and is produced if the default is not null and the default differs from the XSD default for that data type .
- [TYPE.DATATYPE] indicates if property.type.dataType is true.
    - When isDataType is true, [TYPE.NAME] is the name of the XSD built in SimpleType corresponding to property.type, where the prefix is for the xsd namespace.
    - When isDataType is false, [TYPE.NAME] is property.type.name where the tns: prefix is determined by the namespace declaration for the Type's URI. A URI reference to the element containing the DataObject is used and an sdo:propertyType declaration records the target type. Values in XML documents will be of the form "#xpath" where the xpath is an SDO DataObject XPath. It is typical to customize the declaration to IDREF if the target element has an attribute with type customized to ID.
- [OPPOSITE.NAME] is the opposite property if the property is bidirectional and indicated when property.opposite is not null.

SDO Property	**XSD Attribute**
	<xsd:attribute name=[NAME]
[ALIAS_NAME]	sdo:aliasName=[ALIAS_NAME]
[READ_ONLY]	sdo:readOnly=[READ_ONLY]
[DEFAULT]	default=[DEFAULT]
[TYPE.DATATYPE]	type="tns:[TYPE.NAME]"
![TYPE.DATATYPE]	type="xsd:anyURI" sdo:propertyType=tns:[TYPE.NAME]
[OPPOSITE.NAME]	sdo:oppositeProperty=[OPPOSITE.NAME]

Mapping of SDO DataTypes to XSD Built in Data Types

For the SDO Java Types, the corresponding base SDO Type is used. For the SDO Java Types, and for SDO Date, an sdo:dataType annotation is generated on the XML attribute or element referring to the SDO Type.

SDO Type	XSD Type
Boolean	boolean
Byte	byte
Bytes	hexBinary
Character	string
DataObject	anyType
Date	dateTime
DateTime	dateTime
Day	gDay
Decimal	decimal
Double	double
Duration	duration
Float	float
Int	int
Integer	integer
Long	long
Month	gMonth
MonthDay	gMonthDay
Object	anySimpleType
Short	short
String	string
Strings	string
Time	time
Year	gYear
YearMonth	gYearMonth
YearMonthDay	date
URI	anyURI

Example Generated XSD

If the Types and Properties for the PurchaseOrder schema had not come originally from XSD, then these rules would produce the following XML Schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="purchaseOrder" type="PurchaseOrder"/>
 <xsd:complexType name="PurchaseOrder">
  <xsd:sequence>
   <xsd:element name="shipTo" type="USAddress" minOccurs="0"/>
   <xsd:element name="billTo" type="USAddress" minOccurs="0"/>
   <xsd:element name="items"  type="Items" minOccurs="0"/>
```

```
  </xsd:sequence>
  <xsd:attribute name="comment" type="xsd:string"/>
  <xsd:attribute name="orderDate" type="xsd:date"/>
 </xsd:complexType>

 <xsd:element name="uSAddress" type="USAddress"/>
 <xsd:complexType name="USAddress">
  <xsd:attribute name="name"   type="xsd:string"/>
  <xsd:attribute name="street" type="xsd:string"/>
  <xsd:attribute name="city"   type="xsd:string"/>
  <xsd:attribute name="state"  type="xsd:string"/>
  <xsd:attribute name="zip"    type="xsd:decimal"/>
  <xsd:attribute name="country" type="xsd:string" default="US"/>
 </xsd:complexType>

 <xsd:element name="items" type="Items"/>
 <xsd:complexType name="Items">
  <xsd:sequence>
   <xsd:element name="item" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>

 <xsd:element name="item" type="Item"/>
 <xsd:complexType name="Item">
  <xsd:attribute name="productName" type="xsd:string"/>
  <xsd:attribute name="quantity" type="quantityType">
  <xsd:attribute name="partNum"  type="SKU"/>
  <xsd:attribute name="USPrice"  type="xsd:decimal"/>
  <xsd:attribute name="comment"  type=xsd:string"/>
  <xsd:attribute name="shipDate" type="xsd:date"/>
 </xsd:complexType>

 <xsd:simpleType name="quantityType">
  <xsd:restriction base="xsd:int"/>
 </xsd:simpleType>

 <xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string"/>
 </xsd:simpleType>

</xsd:schema>
```

The following is the serialization of the example purchase order that matches this schema.

```
<?xml version="1.0"?>
<purchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    orderDate="1999-10-20" comment="Hurry, my lawn is going wild!">
  <shipTo country="US" name="Alice Smith" street="123 Maple Street"
```

```
    city="Mill Valley" state="CA" zip="90952"/>
  <billTo country="US" name="Robert Smith" street="8 Oak Avenue"
    city="Old Town" state="PA" zip="95819"/>
  <items>
    <item partNum="872-AA" productName="Lawnmower"
      quantity="1" USPrice="148.95" comment="Confirm this is electric"/>
    <item partNum="926-AA" productName="Baby Monitor"
      quantity="1" USPrice="39.98" shipDate="1999-05-21"/>
  </items>
</purchaseOrder>
```

Customizing Generated XSDs

Because an XSD contains more information than Type and Property, there are many XSD capabilites unused by the default generation, for example the preference between serializing with XML elements or attributes. The recommended procedure is to generate the XSD from Types and Properties, customize the XSD using tools or with XSLT, and use the customized XSD as the original from which to define the SDO Types and Properties.

DataGraph XML Serialization

A DataGraph may be serialized as an XML stream. If the Types and Properties came from XML Schema, the DataObjects are serialized following the XSD. If the metadata comes from another source, a virtual SDO XSD is generated and the DataObjects are serialized following the XSD.

The DataGraph's rootObject is a DataObject with exactly one property set. The name of this property is the root element name. The value of this property is the DataObject serialized in the root element. The DataGraph and the rootObject are two extra objects when compared to using DataObjects and XMLHelper. For example, for the purchase order XSD, a DataGraph's rootObject is a document DataObject with a containment property called "purchaseOrder" that contains the actual purchase order DataObject. Applications that do not use DataGraphs just use the purchase order DataObject directly and do not create the DataGraph and the document DataObject.

In general, the DataGraph serialization consists of a description of the schema used for the DataGraph, followed by the DataObjects that are contained in the DataGraph, followed by a description of the changes. The serialization of DataObjects follows the XMI specification or the XSD for the DataObject model, producing the same XML stream independent of the enclosing DataGraph element. When XML Schema is used as the metadata, the XML serialization of the DataObjects follows the XSD and the resulting XML elements should validate with the XML Schema when all the constraints for the XSD are enforced.

The description of the schema is optional and can be expressed either as an XSD or EMOF model. The description of the changes is also optional. The changes are expressed as a change summary. XSDs and models are typically included if it is likely that the reader of the DataGraph would not be able to retrieve the model by the logical URI of the XSD targetNamespace or EMOF Package URI. The serialization of the EMOF models follows the XMI specification. The optional serialization of the ChangeSummary also follows XMI, where properties that have not changed value are omitted. When serializing XSDs and models, only the XSDs and models actually used by the DataObjects are typically transferred. When the DataGraph was originally created from an XSD, the XSD form is preferred in order to preserve all original XSD information. If the DataGraph is from a source other than XSD, an XSD may be generated (typically following the EMOF and XMI specifications) and included, or the EMOF model may be included. The choice of which to include is determined by the serializer of the DataGraph.

The serialization of a DataGraph, whether invoked through a DAS or java.io.Serializable or in a Web service, is expected to be the same XML format described here. When a DataGraph is serialized in Java serialization, it is preceded by an int indicating the number of bytes in the DataGraph XML. When a single DataObject from a DataGraph is serialized, the format is an XPath subset of the DataObject’s path location within the DataGraph from the root, preceded by an int for the number of bytes in the XPath, and followed by the serialization of the DataGraph.

The XSD for the DataGraph serialization is:

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sdo="commonj.sdo"
  targetNamespace="commonj.sdo">

  <xsd:element name="datagraph" type="sdo:DataGraphType"/>

  <xsd:complexType name="DataGraphType">
    <xsd:complexContent>
      <xsd:extension base="sdo:BaseDataGraphType">
        <xsd:sequence>
          <xsd:any minOccurs="0" maxOccurs="1" namespace="##other"
processContents="lax"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="BaseDataGraphType" abstract="true">
    <xsd:sequence>
      <xsd:element name="models" type="sdo:ModelsType" minOccurs="0"/>
      <xsd:element name="xsd" type="sdo:XSDType" minOccurs="0"/>
      <xsd:element name="changeSummary" type="sdo:ChangeSummaryType"
minOccurs="0"/>
    </xsd:sequence>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
  </xsd:complexType>

  <xsd:complexType name="ModelsType">
    <xsd:annotation>
      <xsd:documentation>
        Expected type is emof:Package.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:any minOccurs="0" maxOccurs="unbounded" namespace="##other"
processContents="lax"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="XSDType">
    <xsd:annotation>
      <xsd:documentation>
        Expected type is xsd:schema.
      </xsd:documentation>
```

```
    </xsd:annotation>
    <xsd:sequence>
      <xsd:any minOccurs="0" maxOccurs="unbounded" namespace="http://
www.w3.org/2001/XMLSchema" processContents="lax"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="ChangeSummaryType">
    <xsd:sequence>
      <xsd:any minOccurs="0" maxOccurs="unbounded" namespace="##any"
processContents="lax"/>
    </xsd:sequence>
    <xsd:attribute name="create" type="xsd:string"/>
    <xsd:attribute name="delete" type="xsd:string"/>
    <xsd:attribute name="logging" type="xsd:boolean"/>
  </xsd:complexType>

  <xsd:attribute name="ref" type="xsd:string"/>

</xsd:schema>
```

Examples of this serialization can be seen in Accessing DataObjects using XPath subset and in Appendix – Complete DataGraph Serialization.

XPath Expression for DataObjects

Many of the accessor methods for DataObjects make use of a String parameter that provides the path that identifies the property to which the method applies.

The XPath expression is an augmented subset of XPath 1.0 [5] with the additional ability to access data using 0 as a base index, a style common throughout C++ programming. The syntax for specifying these paths, is shown here:

```
path ::= (scheme ':')? '/'? (step '/')* step
scheme ::= [^:]+ ':'
step ::= '@'? property
      | property '[' index_from_1 ']'
      | property '.' index_from_0
      | reference '[' attribute '=' value ']'
      | ".."
property ::= NCName      ;; may be simple or complex type
attribute ::= NCName     ;; must be simple type
reference :: NCName      ;; must be DataObject type
index_from_0 ::= Digits
index_from_1 ::= NotZero (Digits)?
value ::= Literal
      | Number
      | Boolean
Literal ::= '"' [^"]* '"'
      | "'" [^']* "'"
Number ::= Digits ('.' Digits?)?
      | '.' Digits
Boolean ::= true
      | false
NotZero ::= [1-9]
Digits ::= [0-9]+

;; leading '/' begins at the root
;; ".." is the containing DataObject, using containment properties
;; Only the last step have an attribute as the property
```

The presence or absence of the @ sign in a path has no meaning. Properties are always matched by name independent of their XML representation.

The scheme is an extension mechanism for supporting additional path expressions in the future. No schema and a scheme of "sdo:" are equivalent, representing this syntax.

For example, consider the Company model described in “Complete Data Graph for Company Example” on page 140. One way to construct an XPath that can be used to access a DataObject contained in another DataObject is to specify the index of the contained DataObject within the appropriate property. For example, given an instance of a Company DataObject called “company” one way to access the Department at index 0 in the “departments” list is:

```
DataObjectPtr department = company->getDataObject("departments.0");
```

Another way to access a contained DataObject is to identify that object by specifying the value of one of the attributes of that object. So, for example, given a Department DataObject called "department", one way to access the Employee where the value of the "SN" attribute is "E0002" is:

```
DataObjectPtr employee =
        department->getDataObject("employees[SN='E0002']");
```

It is also possible to write a path expression that traverses one or more references in order to find the target object. The two accesses shown above can be combined into a single call that gets the Employee using a path expression that starts from the company DataObject, for example

```
DataObjectPtr employee =
     company->getDataObject("departments.0/employees[SN='E0002']");
```

If more than one property shares the same name, only the first is matched by the path expression, using property.name for name matching. If there are alias names assigned, those are also used to match. Also, names including any of the special characters of the syntax (./[]="'@) are not accessible. Each step of the path before the last must return a single DataObject. When the property is a Sequence, the values returned are those of the getValue() accessor.

ChangeSummary XML format

The serialization of the ChangeSummary includes enough information to reconstruct the original information of the DataObjects at the point when logging was turned on. The goal of this format is to provide a simple XML representation that can express the difference between the graph when logging began and ended. The serialization of the state when logging is ended is the complete XML as serialized from XMLHelper and is referred to as the final XML in this section to contrast with the changeSummary XML.

DataObjects which are currently in the data graph, but were not present when logging was started are indicated in the change summary with a create attribute:

```
<changeSummary create="E0004" >
</changeSummary>
...
<employees name="Al Smith" SN="E0004"/>
...
```

Similarly, DataObjects deleted during logging are flagged with the "delete" attribute. In this case the change summary also contains a deep copy of the object which was deleted, as it no longer appears in the data graph. Also, the position in the tree must be recorded, so the departments property is reproduced, where there is an employees element for each employee object. The sdo:ref attribute is used to indicate the corresponding object that is represented in both the changeSummary and the final document. For example, `<employees sdo:ref="E0001"/>` refers to the employee with ID E0001 in the final document, `<employees name="John Jones" SN="E0001"/>`. The example below shows that the deleted employee has ID E0002, is located in the first department at the second position. The first and third employees are unchanged and the fourth employee is added.

```
<sdo:datagraph xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">

  <changeSummary create="E0004" delete="E0002">
    <company sdo:ref="#/company" name="ACME" employeeOfTheMonth="E0002"/>
    <departments sdo:ref="#/company/departments[1]">
      <employees sdo:ref="E0001"/>
      <employees name="Mary Smith" SN="E0002" manager="true"/>
      <employees sdo:ref="E0003"/>
    </departments>
  </changeSummary>

  <company:company name="MegaCorp" employeeOfTheMonth="E0004">
    <departments name="Advanced Technologies" location="NY" number="123">
      <employees name="John Jones" SN="E0001"/>
      <employees name="Jane Doe" SN="E0003"/>
      <employees name="Al Smith" SN="E0004" manager="true"/>
    </departments>
  </company:company>
```

```
</sdo:datagraph>
```

The labels above are IDREFs when IDs are available, and SDO XPath expressions otherwise, to locate the data object.

The content of a ChangeSummary element is a deep copy of the objects at the point they were deleted, where the deleted property value was a data object type. The deep copy uses the copy algorithm of the CopyHelper.

Where changes made were only to data type properties, the ChangeSummary element contains copy of the data object from the datagraph, but containing only the properties which have changed, and showing their old values. For example, changing the company name places just the changed information in the change summary.

```
<sdo:datagraph xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">

  <changeSummary>
    <company sdo:ref="#/company" name="ACME"/>
  </changeSummary>

  <company:company name="MegaCorp" employeeOfTheMonth="E0004">
    ...
  </company:company>

</sdo:datagraph>
```

If an old value is not present in the ChangeSummary, it is assumed not to have changed. If the old value was not set, the old value is still represented in the ChangeSummary. The XML for old values of datatype properties doesn't distinguish between not present and default value. For example, if comment is an optional property of product and is set for the first time.

```
<sdo:datagraph xmlns:product="product.xsd"
               xmlns:sdo="commonj.sdo">

  <changeSummary>
    <product sdo:ref="#/product">
      <comment/>
    </product>
  </changeSummary>

  <product:product pid="P123">
    <comment>Sale until the end of the month.</comment>
    ...
  </product:product>

</sdo:datagraph>
```

Multi-valued datatype properties and mutli-valued non-cotnainment properties have their entire old and new values in the changeSummary and final XML respectively. For example, if availableColors is a multi-valued property for a product, and the availableColors change:

```
<sdo:datagraph xmlns:product="product.xsd"
               xmlns:sdo="commonj.sdo">

  <changeSummary>
    <product sdo:ref="#/product">
      <availableColors>blue</availableColors>
      <availableColors>green</availableColors>
    </product>
  </changeSummary>

  <product:product pid="P123">
    <availableColors>blue</availableColors>
    <availableColors>red</availableColors>
    ...
  </product:product>

</sdo:datagraph>
```

Examples

The examples given here assume the use of an XML Data Access Service (XMLDAS) to load and save a data graph from and to XML files. The XMLDAS is referenced here to provide a concrete way of illustrating the objects in the graph and to show the effects of operations on the graph in a standard, easily understood format. The code shown here would work just as well against an equivalent data graph that was provided by any other DAS.

The examples covered here include:

The example model is a Company with a Department containing several Employees. The XSD for the Company is shown in the Appendix, "Complete Data Graph for Company Example" on page 140.

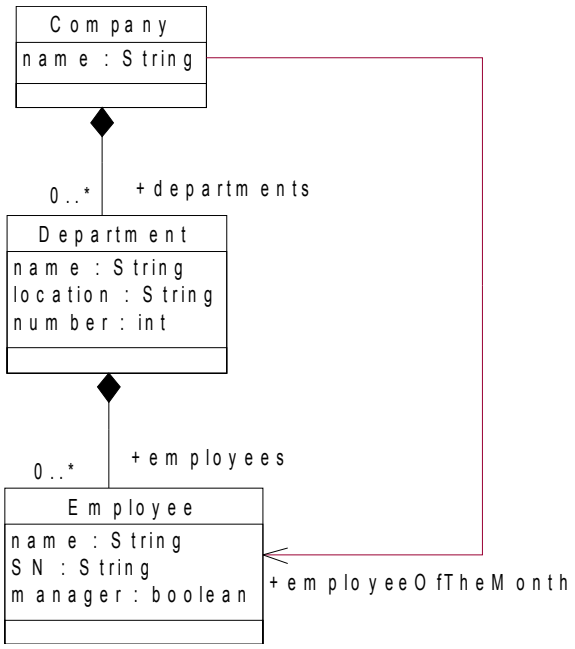


Figure 5: Data Model for Company

Accessing DataObjects using XPath

We can use the XMLHelper to load DataObjects representing a company in a data graph from the following XML file (SN is an XML ID):

```
<sdo:datagraph xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">
  <company:company name="ACME" employeeOfTheMonth="E0002">
    <departments name="Advanced Technologies" location="NY" number="123">
      <employees name="John Jones" SN="E0001"/>
      <employees name="Mary Smith" SN="E0002" manager="true"/>
      <employees name="Jane Doe" SN="E0003"/>
    </departments>
  </company:company>
</sdo:datagraph>
```

(This XML conforms to the company model defined in "Complete Data Graph for Company Example" on page 140.)

The examples show how to use DataObjects and the XMLHelper as well as how to use DataGraph. For DataObjects:

```
// Load and access the company DataObject from
// the "company" property of the data graph.
 DataObjectPtr datagraph = XMLHelper.load(stream).getRootObject();
DataObjectPtr company = datagraph.getDataObject("company");
```

For DataGraph:

```
// Access the company DataObject from the "company" property of
// the root object.
 DataObjectPtr rootObject = dataGraph.getRootObject();
 DataObjectPtr company = rootObject.getDataObject("company");
```

If we wish to change the name of the company DataObject from "ACME" to "MegaCorp", we could use the following:

```
// Set the "name" property for the company
   company->setCString("name", " MegaCorp");
```

Now, suppose we wish to access the employee whose serial number is “E0002”. If we know that this employee is located at index 1 within the department that is located at index 0 within the company object, one way to do this is by traversing each reference in the data graph and locating each DataObject in the many-valued department property using its index in the list. For example, from the company, we can get a list of departments, from that list we can get the department at index 0, from there we can get a list of employees, and from that list we can get the employee at index 1.

```
// Get the list of departments
 DataObjectList& departments = company.getList("departments");
 // Get the department at index 0 on the list
 DataObjectPtr department = departments[0];
 // Get the list of employees for the department
 DataObjectList& employees = department->getList("employees");
 // Get the employee at index 1 on the list
 DataObjectPtr employeeFromList = employees[1];
```

Alternatively, we can write a single XPath expression that directly accesses the employee from the root company.

```
 // Alternatively, an xpath expression can find objects
 // based on positions in lists:
 DataObjectPtr employeeFromXPath =
        company->getDataObject("departments.0/employees.1");
```

Otherwise, if we don’t know the relative positions of the department and employee DataObjects, but we do know that the value number attribute of the department is “123”, we can write an XPath expression that accesses the employee DataObject using the appropriate values:

```
 // Get the same employee using an xpath expression
 // starting from the company
 DataObjectPtr employeeFromXPathByValue = company->getDataObject(
    "departments[number=123]/employees[SN="E0002"]");
```

In order to remove that employee from the data graph, we could use:

```
// remove the employee from the graph
employees.remove(1);
```

And, finally, to create a new employee:

```
// create a new employee

DataObjectPtr newEmployee =
            department->createDataObject("employees");
newEmployee->setCString("name", "Al Smith");
newEmployee->setCString("SN", "E0004");
```

```
  newEmployee->setBoolean("manager", true);

  // Reset employeeOfTheMonth to be the new employee
  company->setDataObject("employeeOfTheMonth", newEmployee);
```

After saving with the XMLHelper, the resulting XML file would contain:

```
xmlHelper.save(datagraph, "commonj.sdo", "datagraph", stream);

<sdo:datagraph xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">

  <changeSummary create="E0004" delete="E0002">
    <company sdo:ref="#/company" name="ACME" employeeOfTheMonth="E0002"/>
    <departments sdo:ref="#/company/departments[1]">
      <employees sdo:ref="E0001"/>
      <employees name="Mary Smith" SN="E0002" manager="true"/>
      <employees sdo:ref="E0003"/>
    </departments>
  </changeSummary>

  <company:company name="MegaCorp" employeeOfTheMonth="E0004">
    <departments name="Advanced Technologies" location="NY" number="123">
      <employees name="John Jones" SN="E0001"/>
      <employees name="Jane Doe" SN="E0003"/>
      <employees name="Al Smith" SN="E0004" manager="true"/>
    </departments>
  </company:company>

</sdo:datagraph>
```

The ChangeSummary provides an overview of the changes that have been made to the data graph. The ChangeSummary contains DataObjects as they appeared prior to any modifications and includes only those objects and properties that have been modified or deleted or which are referenced by a property that was modified. The sdo:ref attribute is used to map DataObjects, in the ChangeSummary, back to the corresponding DataObjects in the data graph.

In this example, the name property of the Company object was changed, so the original company name is shown in the ChangeSummary. However, the name of the Department object was not changed and therefore the department name does not appear. The employees property of the Department object did change (one Employee was added and one Employee was deleted) so the summary includes the list of all the original employees. In the case of the Employee that was

deleted, all the properties are displayed in the summary. Employees that have not changed include the sdo:ref attribute, but the unchanged properties of these employees are not displayed.

All of the DataObjects in this particular example have been affected or referenced by some change, so the ChangeSummary includes references to all of the objects in the original DataGraph. In another situation where only a few DataObjects from a large data graph are modified, the ChangeSummary would include only small subset of the overall data graph.

Note: The serialized data graph can also have optional elements that describe the model and change information. These elements have been omitted in the output shown above. The complete serialization of this data graph is shown in "Complete Data Graph for Company Example" on page 140.

Accessing DataObjects via Property Index

In the previous section, all the fields in a DataObject were specified using XPath strings, where each string was derived from the name of a property. It is also possible to access fields using the index of each property.

The following example has the same effect as the previous example. The indices for the properties are represented as **int** fields. The values are derived from the position of properties as defined in the company.

```
 // Predefine the property indices
int ROOT_COMPANY = 0;

int COMPANY_DEPARTMENT = 0;
int COMPANY_NAME = 1;

int DEPARTMENT_EMPLOYEES = 0;

int EMPLOYEE_NAME = 0;
int EMPLOYEE_SN = 1;
int EMPLOYEE_MANAGER = 2;

 // Load and access the company DataObject from
 // the "company" property of the data graph.
 DataObjectPtr datagraph =xmlHelper.load(stream).getRootObject();
 DataObjectPtr company = datagraph->getDataObject("company");

// Set the "name" property for the company
company->setCString(COMPANY_NAME, "MegaCorp");

// Get the list of departments
```

```
DataObjectList& departments = company->getList(COMPANY_DEPARTMENT);
// Get the department at index 0 on the list
DataObjectPtr department = departments[0];
// Get the list of employees for the department
DataObjectList& employees = department->getList(DEPARTMENT_EMPLOYEES);
// Get the employee at index 1 on the list
DataObjectPtr employeeFromList = employees[1];

// remove the employee from the graph
employees.remove(1);

// create a new employee
DataObjectPtr newEmployee =
               department->createDataObject(DEPARTMENT_EMPLOYEES);
newEmployee->setCString(EMPLOYEE_NAME, "Al Smith");
newEmployee->setCString(EMPLOYEE_SN, "E0004");
newEmployee->setBoolean(EMPLOYEE_MANAGER, true);
```

Accessing the Contents of a Sequence

The following code uses the Sequence interface to analyze the contents of a data graph that conforms to the Letter model. (The definition of this model is shown in the appendix.) This code first goes through the Sequence looking for unformatted text entries and prints them out. Then the code checks to verify that the contents of the "lastName" property of the DataObject matches the contents of the same property of the Sequence:

```
void printSequence(DataObjectPtr letter)
{
  // Access the Sequence of the FormLetter
  SequencePtr letterSequence = letter->getSequence();
  // Print out all the settings that contain unstructured text
  cout << "Unstructured text:" << endl;
  for (int i=0; i<letterSequence.size(); i++)
  {
    if (letterSequence[i].isText())
    {
      cout << "(" << letterSequence.getValue(i) << ")" << endl;
    }
  }

  // Verify that the lastName property of the DataObject has the same
  // value as the lastName property for the Sequence.
  const char* dataObjectLastName = letterDataObject->getCString("lastName");
  for (int i=0; i<letterSequence.size(); i++)
  {
    if (!letterSequence.isText())
```

```
  {
    if (!strcmp("lastName", letterSequence[i].getProperty().getName())
    {
       if (!strcmp(dataObjectLastName,
              letterSequence.getCStringValue(i))
       {
         cout << "Last Name property matches" << endl;
         break;
       }
    }
 }
```

Assume that the following XML file is loaded by the XMLDAS to produce a DataGraph that is passed to the above method:

```
<letter:letters xmlns:letter="http://letterSchema">
  <date>August 1, 2003</date>
  Mutual of Omaha
  Wild Kingdom, USA
  Dear
  <firstName>Casy</firstName>
  <lastName>Crocodile</lastName>
  Please buy more shark repellent.
  Your premium is past due.
</letter:letters>
```

(Note: this XML conforms to the schema defined in "Complete Data Graph for Letter Example" on page 143.)

The output of this method would be:

```
Unstructured text:
  (Mutual of Omaha)
  (Wild Kingdom, USA)
  (Dear)
  (Please buy more shark repellent.)
  (Your premium is past due.)
Last Name property matches
```

Using Type and Property with DataObjects

The Type interface provides access to the metadata for DataObjects in a data graph.

The methods on Type and Property provide information that describes the properties of a DataObject in the data graph. To obtain the Type for a DataObject, use the getType() method.

For example, consider the printDataObject method shown below. This method prints out the contents of a DataObject. Each property is displayed metadata, accessed dynamically, using Type and Property.

```
void printValue(DataObjectPtr dp, const Property& p)
{
  cout << "Single Valued Property:" << p.getName() << endl;
  if (dp->isSet(p)
  {
    cout << p.getType().getName() << "=" << dp->getCString(p) << endl;
  }
  else
  {
    cout << "was unset" << endl;
  }
}

void printList(DataObjectPtr dp, const Property& p)
{
  DataObjectList& dobl = dp->getList(p);
  cout << "Many Valued Property:" << p.getName() << endl;
  if (dobl.size() ==0) {
    cout << "(empty list)" << endl;
    return;
  }

  for (int i=0;i<dobl.size();i++) {
    cout << p.getType().getName() << "[" << i << "]="
         << dobl.getCString(i);
  }
}

void printDataObject(DataObjectPtr dol)
{
  PropertyList pl = dol->getProperties();
  for (int j=0;j< pl.size(); j++)
  {
    if (pl[j].isMany())
    {
      printList(dol,pl[j]);
    }
    else
    {
```

```
    printValue(dol,pl[j]);
   }
  }
}
```

Creating XML from Data Objects

The following code will create and save a purchase order, as shown in the XSD primer. This example makes use of DataFactory and XMLHelper:

```
 DataObjectPtr purchaseOrder =
  myDataFactory.create(null, "PurchaseOrderType");

purchaseOrder->setCString("orderDate", "1999-10-20");

DataObjectPtr shipTo = purchaseOrder->createDataObject("shipTo");
shipTo->setCString("country", "US");
shipTo->setCString("name", "Alice Smith");
shipTo->setCString("street", "123 Maple Street");
shipTo->setCString("city", "Mill Valley");
shipTo->setCString("state", "CA");
shipTo->setCString("zip", "90952");

DataObjectPtr billTo = purchaseOrder.createDataObject("billTo");
billTo->setCString("country", "US");
billTo->setCString("name", "Robert Smith");
billTo->setCString("street", "8 Oak Avenue");
billTo->setCString("city", "Mill Valley");
shipTo->setCString("state", "PA");
billTo->setCString("zip", "95819");
purchaseOrder->setCString("comment", "Hurry, my lawn is going wild!");
DataObjectPtr items = purchaseOrder->createDataObject("items");

DataObjectPtr item1 = items->createDataObject("item");
item1->setCString("partNum", "872-AA");
item1->setCString("productName", "Lawnmower");
item1->setInteger("quantity", 1);
item1->setCString("USPrice", "148.95");
item1->setCString("comment", "Confirm this is electric");

DataObjectptr item2 = items->createDataObject("item");
item2->setCString("partNum", "926-AA");
item2->setCString("productName", "Baby Monitor");
item1->setInteger("quantity", 1);
```

```
    item2->setCString("USPrice", "39.98");
    item2->setCString("shipDate", "1999-05-21");

     XMLHelperPtr xmh = HelperProvider::getXMLHelper(myDataFactory);
     XMLDocumentPtr doc=xmh->createDocument(purchaseOrder,"","purchaseOrder");
     xmh->save(doc,"purchaseorder.xml”);
```

The following output is created:

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
     <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>PA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Mill Valley</city>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

Creating DataObject Trees from XML documents

It is possible to convert to and from XML documents to build DataObject trees, which is useful when assembling DataObjects from several data sources. XMLHelper can be used to do this. For example, suppose the global elements for shipTo and billTo were declared in the PurchaseOrder XSD:

```
<element name="shipTo" type="USAddress"/>
<element name="billTo" type="USAddress"/>
```

To create the shipTo DataObject from XML:

```
const char* shipToXML[1000];

 strcpy(shipToXML, "<shipTo country='US'");
 strcat(shipToXML, "  <name>Alice Smith</name>");
 strcat(shipToXML, "  <street>123 Maple Street</street>");
 strcat(shipToXML, "  <city>Mill Valley</city>");
 strcat(shipToXML, "  <state>PA</state>");
 strcat(shipToXML, "  <zip>90952</zip>");
 strcat(shipToXML, "</shipTo>");
DataObjectPtr shipTo = xmlHelper->load(shipToXML)->getRootObject();
purchaseOrder->setDataObject("shipTo", shipTo);
```

To convert the billTo DataObject to XML:

const char* `billToXML = xmlHelper->save(billTo, null, "billTo");`
`cout <<  billToXML << endl;`

This produces:

```
<?xml version="1.0" encoding="UTF-8"?>
<billTo country="US"
  <name>Robert Smith</name>
  <street>8 Oak Avenue</street>
  <city>Mill Valley</city>
  <zip>95819</zip>
</billTo>
```

Creating open content XML documents

Open content is often used when a DataObject allows new Properties to be used even when they are not known in advance. This occurs often in XML, for example in Web Services where a SOAP envelope is used to wrap contents specific to web service invocations. In the case of SOAP, an Envelope element contains a Body element and the Body element has open content to allow any element inside. This example shows how to make DataObjects for the SOAP Envelope and Body and place inside a Purchase Order.

```
// Create a SOAP envelope and body
```

```
String soap = "http://schemas.xmlsoap.org/wsdl/soap/";
DataObjectPtr envelope = myDataFactory->create(soap, "Envelope");
DataObjectPtr body = envelope->createDataObject("Body");

// The Body is open content.
// Create a purchase order using the XML global element purchaseOrder
const Property& poProperty = xsdHelper->getGlobalProperty(null,
                                                "purchaseOrder", true);
DataObjectPtr po = body->createDataObject(poProperty);

// fill out the rest of the purchase order
po->setDate("orderDate", (SDODate)3567778);
// ...
```

Using the purchase order in a web service and getting the results is straightforward, by invoking the web service and then extracting from the return soap envelope the result purchase order.

```
DataObjectPtr resultEnvelope = WebService.invoke(
   po, "http://werbservices.org/purchaseOrder", soap, "Envelope");

// Get the purchase order from the result envelope
DataObjectPtr resultPo =
   resultEnvelope->getDataObject("Body/purchaseOrder");
```

Web services and DataGraphs Example

Data graphs may be used in Web services by passing the <datagraph> element in the body of a soap message. For example, the data graph in these examples could be included in a soap body sent on the wire in a web service invocation.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/">
  <soap:Header/>
  <soap:Body>
    <sdo:datagraph
      xmlns:company="company.xsd"
      xmlns:sdo="commonj.sdo">
      <company:company name="ACME" employeeOfTheMonth="E0002">
        <departments name="Advanced Technologies" location="NY" number="123">
          <employees name="John Jones" SN="E0001"/>
          <employees name="Mary Smith" SN="E0002" manager="true"/>
          <employees name="Jane Doe" SN="E0003"/>
        </departments>
```

```
      </company:company>
    </sdo:datagraph>
  </soap:Body>
</soap:Envelope>
```

The SDO BaseDataGraphType allows any root DataObject to be included with the “any” element declaration. To constrain the type of root DataObject in DataGraph XML, an extended DataGraph, CompanyDataGraph, can be declared that restricts the type to a single expected kind, CompanyType. The XSD declaration is from the appendix “Complete Data Graph for Company Example” on page 140.

```
<xsd:element name="company" type="company:CompanyType"/>
<xsd:complexType name="CompanyType">
  <xsd:sequence>
    <xsd:element name="departments" type="company:DepartmentType"
maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="employeeOfTheMonth" type="xsd:string"/>
</xsd:complexType>
```

This example shows a companyDataGraph with a CompanyType root DataObject. These XSD declarations define a CompanyDataGraph extending SDO BaseDataGraphType with CompanyType as the type of root DataObject instead of any.

```
<element name="companyDatagraph" type="company:CompanyDataGraphType"/>
<complexType name="CompanyDataGraphType">
  <complexContent>
    <extension base="sdo:BaseDataGraphType">
      <sequence>
        <element name="company" type="company:CompanyType"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

This ensures that only the company element may appear as the root DataObject of the data graph. The SOAP message for the companyDatagraph is:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/">
  <soap:Header/>
  <soap:Body>
    <company:companyDatagraph
      xmlns:company="company.xsd">
      <company:company name="ACME" employeeOfTheMonth="E0002">
```

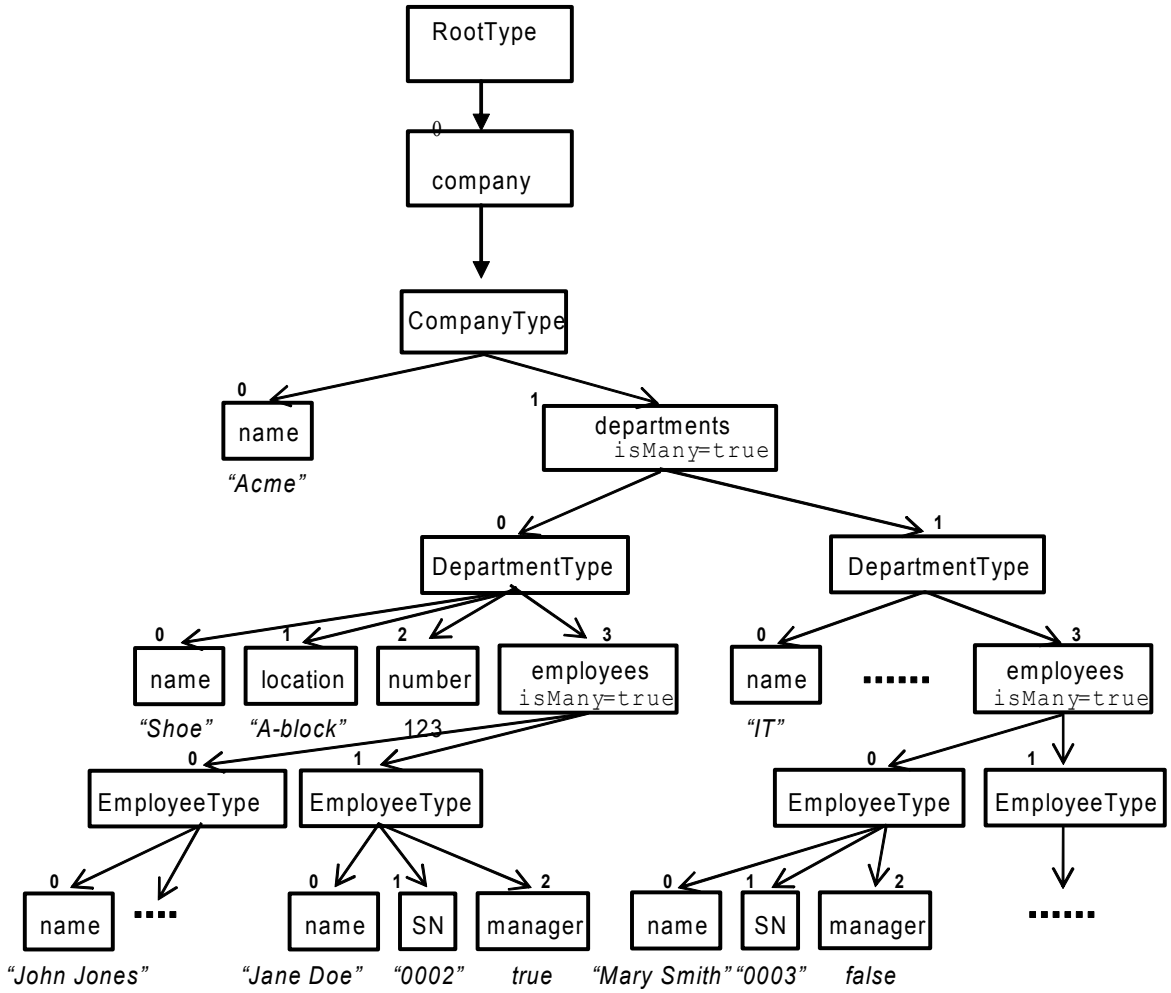
```
        <departments name="Advanced Technologies" location="NY" number="123">
          <employees name="John Jones" SN="E0001"/>
          <employees name="Mary Smith" SN="E0002" manager="true"/>
          <employees name="Jane Doe" SN="E0003"/>
        </departments>
      </company:company>
    </company:companyDatagraph>
  </soap:Body>
</soap:Envelope>
```

The WSDL for the Web service with the companyDatagraph is below. The full listing is shown in the appendix in "Complete WSDL for Web services Example" on page 143.

```
<wsdl:definitions name="Name"
  targetNamespace="http://example.com"
  xmlns:tns="http://example.com"
  xmlns:company="company.xsd"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="company.xsd"
      xmlns:company="company.xsd"
      xmlns:sdo="commonj.sdo"
      elementFormDefault="qualified">
        <element name="companyDatagraph" type="company:CompanyDataGraphType"/>
        <complexType name="CompanyDataGraphType">
          <complexContent>
            <extension base="sdo:BaseDataGraphType">
              <sequence>
                <element name="company" type="company:CompanyType"/>
              </sequence>
            </extension>
          </complexContent>
        </complexType>
    </schema>
  </wsdl:types>
  ...
</wsdl:definitions>
```

Below is an example instance of this data graph (described from the root DataObject down) showing two departments, each with two employees. It shows the data associated with each

property in the data graph, as well as their associated property indices (e.g. **1** for the company's departments property). Use of these indices is demonstrated in the API illustrations.



Setting and Getting Property Values

DataObjects can be navigated using property names:

```
// Assuming we have obtained the root DataObject from a
// data access service
  DataObjectPtr company = rootObject->getDataObject("company");
```

Properties can also be accessed via their property name:

```
const char* companyName = company.getCString("name");
```

Properties can also be accessed via their property index:

```
const char* companyName = company.getCString(0);
```

Properties can be iterated over using a PropertyList. The following iterates over the two company properties:

```
const Type& companyType = company->getType();
PropertyList companyProperties = companyType.getProperties();
for (int i = 0; i < companyProperties.size(); i++)
{
   switch(companyProperties[i].getTypeEnum())
   {
      case Type::BooleanType:
      {
         cout << "Boolean property: "
              << company->getBoolean(i) << endl;
         break;
      }
       ... handle other types
      default:
      {
         cout << "Property as string: "
              << company->getCString(i) << endl;
      }
   }
}
```

Many-valued properties can also be accessed via the List interfaces:

```
DataObjectList depts = company->getDataObjectList("departments");
DataObjectPtr shoeDepartment = depts[0];
```

Many-valued properties can be iterated over using DataObjectLists or primitive arrays:

```
for (int i = 0; i < depts.size(); i++)
  {
     cout << depts.[i].getCString("name") << endl;
  }
```

XPath-like expressions can be used to navigate the properties, the simplest form being the property name:

```
company->setCString("name", "UltraCorp");
```

There are two ways to specify the indices using the XPath support. The following access the same employee:

```
DataObjectPtr jane=
              company->getDataObject("departments.0/employees.1");
DataObjectPtr jane=
              company->getDataObject("departments[1]/employees[2]");
```

Property names are unique per DataObject, so XPath will always return a single property, or throw an error if the property is not found.

Adding New Data

New DataObjects and properties can be added, provided their addition conforms to the model for the SDO.

The following creates and populates a new employee in the IT department:

```
DataObjectPtr itDept =
             company->getDataObject("departments[name="IT"]");
DataObjectPtr newHire = itDept->createDataObject("employees");
newHire->setCString("name", "John Johnson");
newHire->setInteger("SN", 5);
newHire->setBoolean("manager", false);
```

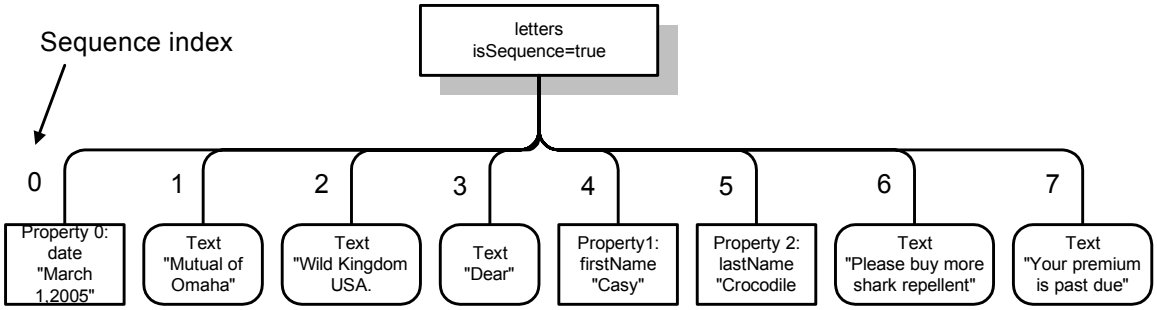
Letter Example

The following shows an XML schema and instance document for a Letter and is used to illustrate the use of the Sequence interface.

```
<sdo:data graph xmi:version="2.0"
  xmlns:xmi="http:www.omg.org/XMI"
  xmlns:sdo="commonj.sdo"
       xmlns:xsd="http:www.w3.org/2001/XMLSchema"
  xmlns:letter="http:letterSchema">
  <xsd>
    <xsd:schema targetNamespace="letter.xsd">
      <xsd:element name="letters" type="letter:FormLetter"/>
      <xsd:complexType name="FormLetter" mixed="true">
        <xsd:sequence>
          <xsd:element name="date" minOccurs="0" type="xsd:string"/>
          <xsd:element name="firstName" minOccurs="0" type="xsd:string"/>
          <xsd:element name="lastName" minOccurs="0" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </xsd>
  <letter:letters>
    <date>March 1, 2005</date>
    Mutual of Omaha
    Wild Kingdom, USA
    Dear
    <firstName>Casy</firstName>
    <lastName>Crocodile</lastName>
    Please buy more shark repellent.
    Your premium is past due.
  </letter:letters>
```

```
</sdo:data graph>
```

Another way to view this instance is as follows:



The letters DataObject contains three properties; date, firstName and lastName. Interspersed among these are two instances of annotation text. These do not have an associated property. The Sequence properties can be accessed using the same mechanisms described in the Company example.

In addition to the property index, Sequences also hold an index into the individual data instances, including annotation text (shown in the diagram by the number in the instance data – e.g. **4**:*"Casy"*). The Sequence index ordering corresponds to the order in which the data was added, rather than any order specified in the model.

The Sequence interface is used to manipulate an DataObject's data when the Sequence index is to be used or preserved.

Setting and Getting Sequence Values

The following accesses the first piece of annotation text (at index 0):

```
SequencePtr lettersSeq = letters->getSequence();
const char* text = lettersSeq->getCString(0);
```

The following both set the last name; the first through the property index, and the second through the sequence index. Setting through the property index will mean the sequence index is not captured.

```
letters->setCString(2, "Smith");  // The index is the property index
lettersSeq->setCStringValue(5, "Smith");// The index is the sequence index
```

Sequences can be iterated over, as shown in the following example. Note the use of getTypeEnum() which returns the type of the property and is used to determine the correct getXXX method to call.

```
for (int i = 0; i < lettersSeq.size(); i++)
{
   switch(lettersSeq->getTypeEnum(i])
   {
      case Type::BooleanType:
      {
         cout << "Boolean property: "
              << lettersSeq->getBooleanValue(i) << endl;
         break;
      }
       ... handle other types
      default:
      {
         cout << "Property as string: "
              << lettersSeq->getCStringValue(i) << endl;
      }
   }
}
```

Adding to a Sequence

When adding new values to a sequence, rather than changing existing values, three pieces of information are required; the sequence index (can default to next available), property index or name, and value.

The following examples add another lastName property (would only be valid if lastName was a many-value property) in the next available sequence index:

```
lettersSeq->addCString("lastName", "Smith");
lettersSeq->addCString(2, "Smith");            // Uses property index
```

Annotation text can be added:

```
lettersSeq->addText("Cancel Subscription.");
```

New values can be added at a specific sequence index using one of the addXXX methods with a first parameter of the index for insertion, and result in entries at later positions being shifted upwards:

```
lettersSeq->addText(1, "Care of");
```

Examples using SDO SPI

A data access service will use the SDO SPI described in section 7 to create the Types and Properties for a model. The following examples show an extract of how a data meditator service would construct the model for the company example described in section 7.2.

The client would construct a new instance of a data access service, passing information about the model to be used. For example, a client could construct an XML data access service and pass a reference to an XML schema that would be used to create the Types and Properties. To create properties and types the data access service must get a DataFactory.

```
DataFactoryPtr df = DataFactory::getDataFactory();
```

The data access service can then create Types and Properties:

```
// create the root type
df->addType("xmldas","RootType");

// create the company type
df->addType("companyNS","CompanyType");

// add the company to the root type
df->addPropertyToType(
   "xmldas", "RootType",     // property goes on this type
   "company",                // name of property
   "companyNS","CompanyType",// type of property
   false,                  // multi-valued
   false,                  // read only
   true);                  // containment

// add the department type
df->addType("companyNS","DepartmentType");

// add the department to the company
df->addPropertyToType(
   "companyNS", "CompanyType",  // property goes on this type
   "departments",               // name of property
   "companyNS","DepartmentType",// type of property
   true,                      // multi-valued
   false,                     // read only
   true);                     // containment
```

Once the data access service has created the model, it can return to the client. The client can then call the data access service to get a data graph. The data access service should use the same data factory to create the instance data.

```
// create an instance of the root
DataObjectPtr root = df->create("xmldas", "RootType");

// create the company
DataObjectPtr company = root->createDataObject("company");
```

Complete Data Graph Examples

Complete Data Graph Serialization

As mentioned in the section on Data Graph Serialization, the serialization of a data graph includes optional elements, that describe the model and the change information, in addition to the DataObjects in the data graph.

The model may be described either as an instance of an XML Schema or EMOF Package (See "Complete Data Graph for Company Example" on page 140) or using an XML Schema (see "Complete Data Graph for Letter Example" on page 143.)

Complete Data Graph for Company Example

The following XML represents the complete serialization of the data graph that includes the changes from the processing described in "Accessing DataObjects using XPath" on page 121.

```
<sdo:datagraph xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">
  <xsd>
    <xsd:schema targetNamespace="company.xsd">
      <xsd:element name="company" type="company:CompanyType"/>
      <xsd:complexType name="CompanyType">
        <xsd:sequence>
          <xsd:element name="departments" type="company:DepartmentType"
maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="employeeOfTheMonth" type="xsd:string"/>
      </xsd:complexType>
      <xsd:complexType name="DepartmentType">
        <xsd:sequence>
          <xsd:element name="employees" type="company:EmployeeType"
maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="location" type="xsd:string"/>
        <xsd:attribute name="number" type="xsd:int"/>
      </xsd:complexType>
      <xsd:complexType name="EmployeeType">
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="SN" type="xsd:ID"/>
        <xsd:attribute name="manager" type="xsd:boolean"/>
```

```
      </xsd:complexType>
    </xsd:schema>
  </xsd>
  <changeSummary create="E0004" delete="E0002">
    <company sdo:ref="#/company" name="ACME" employeeOfTheMonth="E0002"/>
    <departments sdo:ref="#/company/departments[1]">
      <employees sdo:ref="E0001"/>
      <employees name="Mary Smith" SN="E0002" manager="true"/>
      <employees sdo:ref="E0003"/>
    </departments>
  </changeSummary>

  <company:company name="MegaCorp" employeeOfTheMonth="E0004">
    <departments name="Advanced Technologies" location="NY" number="123">
      <employees name="John Jones" SN="E0001"/>
      <employees name="Jane Doe" SN="E0003"/>
      <employees name="Al Smith" SN="E0004" manager="true"/>
    </departments>
  </company:company>
</sdo:datagraph>
```

When using EMOF as metadata, the complete data graph serialization is:

```
<sdo:datagraph xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:company="company.xsd"
               xmlns:emof="http://schema.omg.org/spec/mof/2.0/emof.xmi"
               xmlns:sdo="commonj.sdo">
  <models>
    <emof:Package name="companyPackage"
                  uri="companySchema.emof">
      <ownedType xsi:type="emof:Class" name="CompanySchema">
        <ownedProperty name="company" type="#model.0" containment="true"/>
      </ownedType>
      <ownedType xsi:type="emof:Class" xmi:id="model.0" name="Company">
        <ownedProperty name="departments" type="#model.1" upperBound="-1"
                       containment="true"/>
        <ownedProperty name="employeeOfTheMonth" type="#model.7"/>
        <ownedProperty name="name">
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#String"/>
        </ownedProperty>
      </ownedType>
      <ownedType xsi:type="emof:Class" xmi:id="model.1" name="Department">
        <ownedProperty name="employees" type="#model.2" upperBound="-1"
                       containment="true"/>
        <ownedProperty name="name">
              <type xsi:type="emof:DataType"
```

```
                href="http://schema.omg.org/spec/mof/2.0/emof.xmi#String"/>
        </ownedProperty>
        <ownedProperty name="location" >
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#String"/>
        </ownedProperty>
        <ownedProperty name="number" >
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#Integer"/>
        </ownedProperty>
      </ownedType>
      <ownedType xsi:type="emof:Class" xmi:id="model.2" name="Employee">
        <ownedProperty name="name">
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#String"/>
        </ownedProperty>
        <ownedProperty name="SN">
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#String"/>
        </ownedProperty>
        <ownedProperty name="manager">
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#Boolean"/>
        </ownedProperty>
        <ownedProperty name="employeeStatus" type="#model.3"/>
      </ownedType>
      <ownedType xsi:type="emof:Enumeration" xmi:id="model.3">
        <ownedLiteral name="fullTime" value="1"/>
        <ownedLiteral name="partTime" value="2"/>
      </ownedType>
    </emof:Package>
  </models>
  <changeSummary create="#id.4" delete="#log.0">
    <company sdo:ref="#id.0" name="ACME" employeeOfTheMonth="#log.0"/>
    <departments sdo:ref="#id.1">
      <employees sdo:ref="#id.2"/>
      <employees xmi:id="log.0" name="Mary Smith" SN="E0002" manager="true"/>
      <employees sdo:ref="#id.3"/>
    </departments>
  </changeSummary>
  <company:company xmi:id="id.0" name="MegaCorp" employeeOfTheMonth="#id.4">
    <departments xmi:id="id.1" name="Advanced Technologies" location="NY"
number="123">
      <employees xmi:id="id.2" name="John Jones" SN="E0001"/>
      <employees xmi:id="id.3" name="Jane Doe" SN="E0003"/>
      <employees xmi:id="id.4" name="Al Smith" SN="E0004" manager="true"/>
    </departments>
  </company:company>
</sdo:datagraph>
```

Complete Data Graph for Letter Example

This data graph is used as the input for the example shown in "Accessing the Contents of a Sequence " on page 125. In this case the XSD for the letter is sent as an option, along with the DataObjects. No summary information is sent. When the receiver reads the data graph, the XSD is the metadata and the letter is the data.

```
<sdo:datagraph xmlns:sdo="commonj.sdo"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:letter="http://letterSchema">
  <xsd>
    <xsd:schema targetNamespace="letter.xsd">
      <xsd:element name="letters" type="letter:FormLetter"/>
      <xsd:complexType name="FormLetter" mixed="true">
        <xsd:sequence>
          <xsd:element name="date" minOccurs="0" type="xsd:string"/>
          <xsd:element name="firstName" minOccurs="0" type="xsd:string"/>
          <xsd:element name="lastName" minOccurs="0" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </xsd>
  <letter:letters>
    <date>August 1, 2003</date>
    Mutual of Omaha
    Wild Kingdom, USA
    Dear
    <firstName>Casy</firstName>
    <lastName>Crocodile</lastName>
    Please buy more shark repellent.
    Your premium is past due.
  </letter:letters>
</sdo:datagraph>
```

Complete WSDL for Web services Example

The full WSDL from the Using Web services with data graph Example.

```
<wsdl:definitions name="Name"
  targetNamespace="http://example.com"
  xmlns:tns="http://example.com"
  xmlns:company="company.xsd"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
  <wsdl:types>
   <schema xmlns="http://www.w3.org/2001/XMLSchema"
     targetNamespace="company.xsd"
     xmlns:company="company.xsd"
     xmlns:sdo="commonj.sdo"
     elementFormDefault="qualified">
       <element name="companyDatagraph" type="company:CompanyDataGraphType"/>
       <complexType name="CompanyDataGraphType">
         <complexContent>
           <extension base="sdo:BaseDataGraphType">
             <sequence>
               <element name="company" type="company:CompanyType"/>
             </sequence>
           </extension>
         </complexContent>
       </complexType>
   </schema>
  </wsdl:types>
  <wsdl:message name="fooMessage">
    <wsdl:part name="body" element="company:companyDataGraph"/>
  </wsdl:message>
  <wsdl:message name="fooResponseMessage"></wsdl:message>
  <wsdl:portType name="fooPortType">
    <wsdl:operation name="myOperation">
     <wsdl:input message="tns:fooMessage"/>
     <wsdl:output message="tns:fooResponseMessage"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="fooBinding" type="tns:fooPortType">
    <soap:binding style="document"
     transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="myOperation">
     <soap:operation/>
     <wsdl:input>
      <soap:body use="literal"/>
     </wsdl:input>
     <wsdl:output>
      <soap:body use="literal"/>
     </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="myService">
    <wsdl:port name="myPort" binding="tns:fooBinding">
     <soap:address location="http://localhost/myservice"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

DataType Conversions

The SDO specification defines a set of Abstract SDO Types in the “commonj.sdo” namespace.The following table shows the mapping between the SDO Abstract Types and the C++ types that will be used to represent the values. The TypeEnum values

SDO Abstract Type	C++SDO Type	C++ Type	Comments
Boolean	Type::BooleanType	bool	
BigDecimal	Type::BigDecimalType	wchar_t*/char*	
BigInteger	Type::BigIntegerType	wchar_t*/char*	
Byte	Type::ByteType	char	Single character.
Bytes	Type::BytesType	char *	Array of single characters.
Character	Type::CharacterType	wchar_t	Wide Character.
Date	Type::DateType	SDODate	The C runtime library date/time
Double	Type:: DoubleType	long double	Where available in the C++ library, otherwise double. Size will depend on platform.
Float	Type::FloatType	float	Minimum of 32bit
Integer	Type::IntegerType	long	
Long	Type::LongType	int64_t	__int64 in windows
Object			Not supported
Short	Type::ShortType	short	C++ is typically 32 bit signed (platform dependent).
String	Type::StringType	wchar_t*/char*	
URI	Type::StringType	wchar_t* /char*	

Note: Strings may be retrieved using getString() which returns a const char* pointer to a null terminated string, or getString() , which fills a supplied buffer with a non null-terminated array of wide characters.

Type conversions.

The following conversions will be supported when getting a value of a property.

SDO Abstract Type	DataObject method and return type.	Comments
BigDecimal (Types::BigDecimal)	bool getBoolean()	True if value is non-zero
	char getByte()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	unsigned int getBytes(.. char* buffer)	An array of bytes representing the bigdecimal
	wchar_t getCharacter()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	unsigned int getString(.. Wchar_t* buffer)	An array of wide chars representing the bigdecimal
	short getShort()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	SDODateSDODate getDate()	SDOInvalidConversionException
	long getInteger()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	int64_t getLong()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	float getFloat()	Possible loss of precision. Original value might be out of range, in which case result is undefined.
	long double getDouble()	Possible loss of precision. Original value might be out of range, in which case result is undefined
	const char* getCString()	A null terminated string of bytes representing the bigdecimal

SDO Abstract Type	DataObject method and return type.	Comments
BigInteger (Types::BigInteger)	Bool getBoolean()	True if value is non-zero
	Char getByte()	Original value might be out of range, in which case result is undefined
	unsigned int getBytes(.. char* buffer)	An array of wide chars representing the biginteger
	wchar_t getCharacter()	Original value might be out of range, in which case result is undefined
	unsigned int getString(.. Wchar_t* buffer)	An array of wide chars representing the biginteger
	short getShort()	Original value might be out of range, in which case result is undefined
	SDODate getDate()	SDOInvalidConversionException
	long getInteger()	Original value might be out of range, in which case result is undefined
	int64_t getLong()	Original value might be out of range, in which case result is undefined
	float getFloat()	Original value might be out of range, in which case result is undefined
	long double getDouble()	Original value might be out of range, in which case result is undefined
	const char* getCString()	A null terminated string of bytes representing the biginteger

SDO Abstract Type	DataObject method and return type.	Comments
Boolean (Type::Boolean)	bool getBoolean()	
	char getByte()	0 or 1
	unsigned int getBytes(.. char* buffer)	string "true" if the boolean is true, otherwise "false"
	wchar_t getCharacter()	0 or 1
	unsigned int getString(.. Wchar_t* buffer)	string "true" if the boolean is true, otherwise "false"
	short getShort()	0 or 1
	long getInteger()	0 or 1
	int64_t getLong()	0 or 1
	float getFloat()	0.0 or 1.0
	long double getDouble()	0.0 or 1.0
	const char* getCString()	string "true" if the boolean is true, otherwise "false"

SDO Abstract Type	DataObject method and return type.	Comments
Byte (Types::Char)	bool getBoolean()	true if the byte is non zero, otherwise false
	char getByte()	
	unsigned int getBytes(.. char* buffer)	
	wchar_t getCharacter()	
	unsigned int getString(.. Wchar_t* buffer)	
	short getShort()	
	long getInteger()	
	int64_t getLong()	
	float getFloat()	
	long double getDouble()	
	const char* getCString()	A null terminated string

SDO Abstract Type	DataObject method and return type.	Comments
Bytes (Types::Chars)	bool getBoolean()	true if the bytes contain "true", otherwise false
	char getByte()	SDOInvalidConversionException
	unsigned int getBytes(.. char* buffer)	A non null terminated buffer of byes
	wchar_t getCharacter()	SDOInvalidConversionException
	unsigned int getString(.. Wchar_t* buffer)	A non null terminated buffer of wide characters
	short getShort()	result of "atoi" on the bytes.
	long getInteger()	result of "atol" on the bytes.
	int64_t getLong()	result of "atoi64" or "strtoll" on the bytes
	Float getFloat()	result of "atof"
	long double getDouble()	TBD
	const char* getCString()	A null terminated string

SDO Abstract Type	DataObject method and return type.	Comments
Character (Types::WideChar)	bool getBoolean()	true if char != 0
	char getByte()	value of charactetr if within the char range, otherwise zero
	unsigned int getBytes(.. char* buffer)	A non null terminated buffer of two bytes
	wchar_t getCharacter()	
	unsigned int getString(.. Wchar_t* buffer)	A non null terminated buffer of a single wide character
	short getShort()	The value of the character expressed as a short
	long getInteger()	The value of the character expressed as a long
	int64_t getLong()	The value of the character expressed as a long long
	float getFloat()	TBD
	long double getDouble()	TBD
	const char* getCString()	A null terminated string containing the value (E.g "57")

SDO Abstract Type	DataObject method and return type.	Comments
Date (Type::Date)	bool getBoolean()	SDOInvalidConversionException
	char getByte()	SDOInvalidConversionException
	unsigned int getBytes(.. char* buffer)	SDOInvalidConversionException
	wchar_t getCharacter()	SDOInvalidConversionException
	unsigned int getString(.. Wchar_t* buffer)	SDOInvalidConversionException
	short getShort()	SDOInvalidConversionException
	SDODate getDate()	The date as a SDODate.
	long getInteger()	SDOInvalidConversionException
	int64_t getLong()	SDOInvalidConversionException
	float getFloat()	SDOInvalidConversionException
	long double getDouble()	SDOInvalidConversionException
	const char* getCString()	A null terminated string representing the date

SDO Abstract Type	DataObject method and return type.	Comments
Double (Type::Double)	bool getBoolean()	true if the value is non-zero
	char getByte()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	unsigned int getBytes(.. char* buffer)	Loss of fractional part, original value might be out of range, in which case result is undefined.
	wchar_t getCharacter()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	unsigned int getString(.. Wchar_t* buffer)	SDOInvalidConversionException
	short getShort()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	long getInteger()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	int64_t getLong()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	float getFloat()	The value expressed as a float, with possible loss of precision
	long double getDouble()	
	const char* getCString()	A null terminated string representing the the double (“57E+5)

SDO Abstract Type	DataObject method and return type.	Comments
Float (Type::Float)	bool getBoolean()	true if the value is non-zero
	char getByte()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	unsigned int getBytes(.. char* buffer)	Loss of fractional part, original value might be out of range, in which case result is undefined.
	wchar_t getCharacter()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	unsigned int getString(.. Wchar_t* buffer)	SDOInvalidConversionException
	short getShort()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	long getInteger()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	int64_t getLong()	Loss of fractional part, original value might be out of range, in which case result is undefined.
	float getFloat()	
	long double getDouble()	
	const char* getCString()	A null terminated string representing the the float (“57E+5)

SDO Abstract Type	DataObject method and return type.	Comments
Integer (Type::Integer)	bool getBoolean()	true if the value is non-zero
	char getByte()	Original value may be out of range, typically only low-order bytes are copied.
	unsigned int getBytes(.. char* buffer)	
	wchar_t getCharacter()	Original value may be out of range, typically only low-order bytes are copied.
	unsigned int getString(.. Wchar_t* buffer)	
	short getShort()	Original value may be out of range, typically only low-order bytes are copied.
	long getInteger()	
	int64_t getLong()	
	float getFloat()	
	long double getDouble()	
	const char* getCString()	A null terminated string representing the int (“57")

SDO Abstract Type	DataObject method and return type.	Comments
Long (Type::Long)	bool getBoolean()	true if the value is non-zero
	char getByte()	Original value may be out of range, typically only low-order bytes are copied.
	unsigned int getBytes(.. char* buffer)	
	wchar_t getCharacter()	Original value may be out of range, typically only low-order bytes are copied.
	unsigned int getString(.. Wchar_t* buffer)	
	short getShort()	Original value may be out of range, typically only low-order bytes are copied.
	long getInteger()	Original value may be out of range, typically only low-order bytes are copied.
	int64_t getLong()	
	float getFloat()	
	long double getDouble()	
	const char* getCString()	A null terminated string representing the long (“57")

SDO Abstract Type	DataObject method and return type.	Comments
Short (Type::Short)	bool getBoolean()	true if the value is non-zero
	char getByte()	Low order byte of the short
	unsigned int getBytes(.. char* buffer)	
	wchar_t getCharacter()	The short as a wide character
	unsigned int getString(.. Wchar_t* buffer)	
	short getShort()	
	long getInteger()	
	int64_t getLong()	
	float getFloat()	
	long double getDouble()	
	const char* getCString()	A null terminated string representing the short (“57")

SDO Abstract Type	DataObject method and return type.	Comments
String (Type::WideChars)	bool getBoolean()	true if the value is "true"
	char getByte()	The first character of the string.
	unsigned int getBytes(.. char* buffer)	The string as a non null-terminated buffer.
	wchar_t getCharacter()	The first two characters of the string as a wide-char
	unsigned int getString(.. Wchar_t* buffer)	The string as a wide-character buffer.
	short getShort()	The result of atoi on the string
	long getInteger()	The result of atol on the string
	int64_t getLong()	The result of atoi64 or strtoll on the string
	float getFloat()	The result of atof on the string
	long double getDouble()	TBD
	const char* getCString()	A null terminated string

SDO Abstract Type	DataObject method and return type.	Comments
URI(Type::WideChars)	bool getBoolean()	true if the value is "true"
	char getByte()	The first character of the string.
	unsigned int getBytes(.. char* buffer)	The string as a non null-terminated buffer.
	wchar_t getCharacter()	The first two characters of the string as a wide-char
	unsigned int getString(.. Wchar_t* buffer)	The string as a wide-character buffer.
	short getShort()	The result of atoi on the string
	long getInteger()	The result of atol on the string
	int64_t getLong()	The result of atoi64 or strtoll on the string
	float getFloat()	The result of atof on the string
	long double getDouble()	TBD
	const char* getCString()	A null terminated string

The corresponding setter methods will set the values by performing the reverse conversions to those above. String values will be parsed to produce numerical values, with corresponding loss of fractional parts or precision depending on the original contents of the string.

Trademarks

IBM is a registered trademark of International Business Machines Corporation.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

References

[1] EMOF compliance point from Meta Object Facility 2.0 Core Final Submission,
http://www.omg.org/cgi-bin/doc?ad/2003-04-07

[2] XML Schema Part 1: Structures,
http://www.w3.org/TR/xmlschema-1

[3] Next-Generation Data Programming with Service Data Objects
Any one of:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sdo/
- http:/www.sybase.com

[4] MOF2 XMI Final submission
http://www.omg.org/docs/ad/03-04-04.pdf

[5] XPath 1.0 specification
http://www.w3.org/TR/xpath

[6] Java 1.5.0 API documentation
http://java.sun.com/j2se/1.5.0/

[7] XML Schema Part 2: Datatypes
http://www.w3.org/TR/xmlschema-2