Service Data Objects

IBM Corp. and BEA Systems, Inc.
Version 1.0
November 2003

Authors

John Beatty, BEA Systems, Inc.
Stephen Brodsky, IBM Corporation
Raymond Ellersick, IBM Corporation
Martin Nally, IBM Corporation
Rahul Patel, BEA Systems, Inc.

Status of this Document

This specification may change before final release and you are cautioned against relying on the content of this specification. IBM and BEA 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 API for the Java^{TM} platform that unifies data programming across data source types, provides robust support for common application patterns, and enable 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].

The core concepts in the SDO architecture are the *Data Object* and *Data Graph*. A Data Object holds a set of named properties, each of which contains either a primitive-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 also have the responsibility to track changes made to the graph of Data Objects, including inserts, deletes, and modification to Data Object properties.

Data Graphs are typically constructed from data sources, such as XML files, EJBs, XML databases, relational databases, or from services, such as Web services, JCA Resource Adapters, JMS messages, etc. Components that can populate Data Graphs from data sources and commit changes to Data Graphs back to the data source are called *data mediator services (DMS)*. DMS architecture and APIs are outside the scope of this specification.

Requirements

The scope of the SDO specification includes the following requirements:

- **Dynamic Data API.** Data Objects commonly have typed Java interfaces. However, sometimes it is either impossible or undesirable to create Java interfaces to represent the Data Objects. One common situation in which this occurs is where the data being transferred is defined by the output of a query. Examples would be a relational query against a relational persistence store, or EJBQL queries against an EJB entity bean domain model, Web services, or XML Query queries against an XML source. 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.

- **Support for Static Data API.** In cases where metadata is known at development time (e.g., 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, JCA connectors, JMS message formats, UML models, etc. 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.

- **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 and removes, and reordering have to be tracked as well as simple changes. Service Data Objects will support arbitrary graphs of Data Objects with full change summaries.

- **Change History.** It is a common pattern for a client to receive a Data Object from another tier, make updates to the Data Object, and then pass the modified Data Object back to the other tier. To support this scenario, it is often important for the tier receiving the modified Data Object to know what modifications were made. In simple cases, knowing whether or not the Data Object was modified may be enough. For other cases, it may 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 will support full change history.

- **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 XPath 1.0 expressions.

- **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. As Java reflection does not return sufficient information, SDO provides APIs for metadata. SDO metadata may be derived from XML Schema, EMOF, Java, relational databases, and other structured representations.

- **Validation and Constraints.** SDO supports validation of the standard set of constraints captured in the metadata, which captures common constraints expressible in XML Schema and relational models (e.g., occurrence constraints). SDO also provides an extensibility mechanism for adding custom constraints and validation.

- **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 the value of the department is set for an employee, the employee should be automatically removed from the list of employees help by the department it is currently in, and 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.

The following areas are out of scope:

Complete metamodel and metadata API. SDO includes a minimal metadata access API for use by Data Object client programmers. The intent 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 Essential Meta Object Facility (EMOF) compliance point from the MOF2 specification [2]. Java annotations in JSR 175 may be a future source of metadata.

- **Data Mediator Service specification.** Service Data Objects can be used in conjunction with “mediators,” which 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 mediator framework is out of scope but could be included in a future specification.

Organization of this Document

This specification is organized as follows:

- **Architecture**: Describes the overall SDO system.
- **Java API**: Defines and describes the Java API for SDO.
- **DataGraph Serialization Specification**: Defines how Data Graphs are serialized as XML.
- **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.

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 `DataGraph` is used to collect a graph of related `DataObject`s. The `DataGraph` keeps track of the schema that describes the `DataObject`s. The `DataGraph` also maintains a `ChangeSummary`, which represents the changes made to the `DataObject`s in the graph.

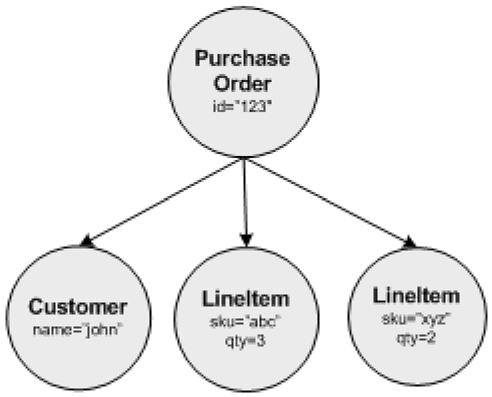

Figure 1: Data Graph containing Data Objects

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

DMS typically use a disconnected data architecture, whereby the client remains disconnected from the DMS except when reading a `DataGraph` or writing back a `DataGraph`. Thus, a typical scenario for using a `DataGraph` involves the following steps:

1. The end user sends a request to a DMS to load a `DataGraph`

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

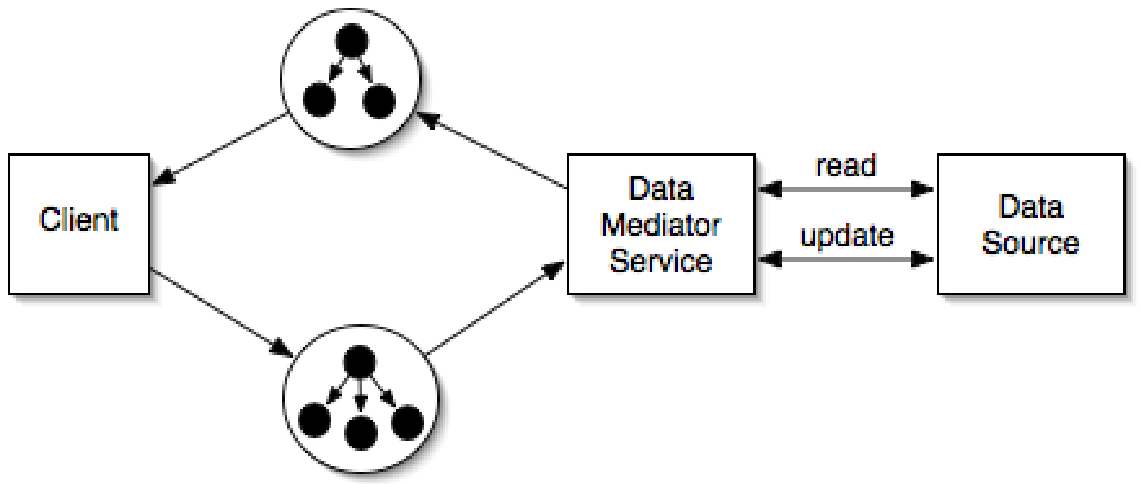

Figure 2: SDO's disconnected data architecture

Note that there are two distinct roles that can be identified among `DataObject` users: the client and the DMS writer. The client needs to be able to traverse a `DataGraph` to access each `DataObject` contained in that `DataGraph` and to get and set the fields in each `DataObject`. The client may also need to serialize and deserialize a `DataGraph`. The DMS writer must be able to define a model for a `DataGraph`, create a new `DataGraph`, generate change history information, and access change history information. This specification's focus is the perspective of the client.

Java API

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

- DataObject – A business data object
- Sequence - A sequence of settings
- DataGraph – A graph of `DataObject`s
- ChangeSummary – Summary of changes to the `DataObject`s in a `DataGraph`

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

- Type – The `Type` of a DataObject or Property.
- Property - A `Property` of a Data Object.

The APIs are shown in figure 3 below.

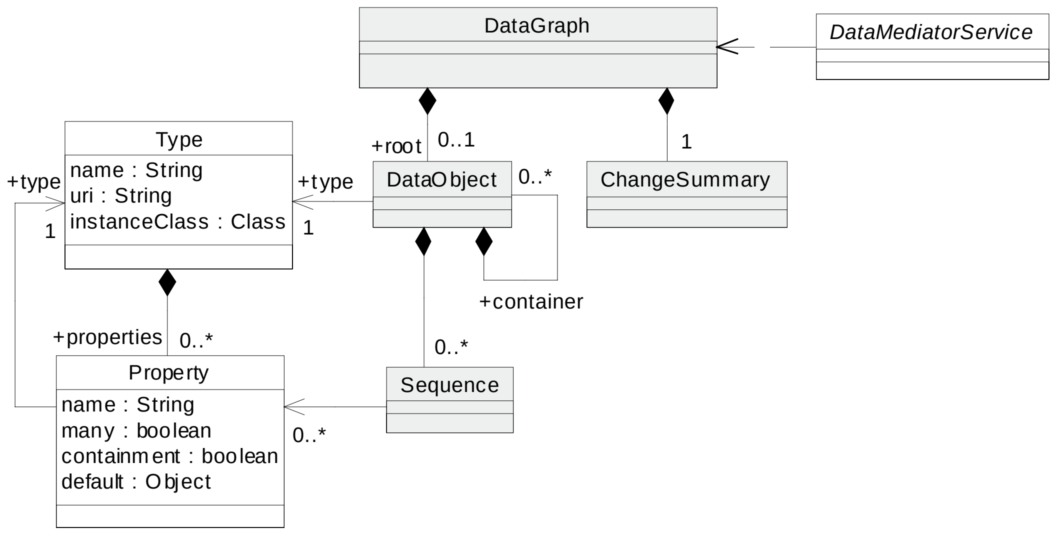

Figure 3: DataGraph Java APIs

DataObject

DataObject is designed to be easy for Java programmers, with one interface the provides access to business data of all the common types and access patterns, such as name, index, and path. The `DataObject` interface provides a rich set of methods that retrieve and update the contents of a `DataObject`. It also provides methods to access the container of the `DataObject and` the `DataGraph` to which the `DataObject` belongs, to create a new instance of a contained `DataObject`, and to delete a `DataObject` from its container. Lastly, `DataObject` provides the ability to get the `DataObject`'s `Type`.

A `DataObject` is composed of properties. Each property can be accessed either by specifying the `Property` object, the property's name, or the property's index. `DataObject`s are linked together by their properties, which means that it is possible to identify a property that belongs to another `DataObject` by specifying a *path* expression that identifies all the links that must be followed to access that other `DataObject`. (More on how to specify a path expression later.)

There are several different accessor methods that can be used to get and set the values of the properties for a given `DataObject`. The choice of which accessor method to use to get or set a particular property depends on:

- Whether the property is specified using an index or a path.
- The type of the property.
- Whether the property is single-valued or many-valued.

When a `DataObject`'s typed accessors get<T>() and set<T>()are invoked, a type conversion is necessary if the value is not already an instance of the requested type T. 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 on `DataObject`, with possible loss of information. The supported data type set is initially the Java primitives, object wrappers of Java primitives, String, Date, byte[], BigDecimal, and BigInteger, with conversions specified in Java [6]. These initial data types may be expanded to include the expected XML types from JAXP 1.3 when it becomes available.

The isSet(property) accessors return true when the value of the property is set to a value other than the default value. The unset(property) accessor resets the property, so that isSet(property) returns false and get(property) returns the default. Unset may be thought of as clearing out a single property, while delete() unsets all the properties. If a property uses settings (settings are described in `Sequence`), there is a distinction between set(property, property.getDefault()) and unset(property). For properties that use settings, after set(property, property.getDefault()), isSet(property) returns true. For properties that do not use settings, after set(property, property.getDefault()), isSet(property) returns false. After unset(property) and delete(), isSet(property) always returns false.

Get(property) with property.isMany() true always return a List. The getList(property) accessor is especially convenient for many-valued properties. The set(property, value) and setList(property, value) accessors for property.isMany() true require that value be a java.util.Collection and List respectively, and are equivalent to getList(property).clear() followed by getList(property).addAll(value).

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 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 removes the `DataObject` from its containing `DataObject`, unsets all its properties, and deletes all composed `DataObject`s. One-way non-containment properties in other `DataObject`s referring to deleted `DataObject`s are not modified, but may need to be changed to other values to restore closure to the `DataGraph`. A deleted `DataObject` may be used again, have its values set, and added into the `DataGraph` again.

The `getContainer()` method returns the parent `DataObject` and the `getContainmentProperty()` method returns the `Property` of the container which contains this object, providing simple navigation up and down the `DataObject` containment tree.

To access the `DataGraph` from a `DataObject`, `getDataGraph()` is used.

The `Type` of the `DataObject` is returned by `getType()`.

The ability to obtain a java.util.Map for a `DataObject` is part of the SDO roadmap.

```
public interface DataObject extends Serializable
{
  Object get(String path);
  void set(String path, Object value);
  boolean isSet(String path);
  void unset(String path);

  boolean getBoolean(String path);
  byte getByte(String path);
  char getChar(String path);
  double getDouble(String path);
  float getFloat(String path);
  int getInt(String path);
  long getLong(String path);
  short getShort(String path);
  byte[] getBytes(String path);
  BigDecimal getBigDecimal(String path);
  BigInteger getBigInteger(String path);
  DataObject getDataObject(String path);
  Date getDate(String path);
  String getString(String path);
  List getList(String path);
  Sequence getSequence(String path);

  void setBoolean(String path, boolean value);
  void setByte(String path, byte value);
  void setChar(String path, char value);
  void setDouble(String path, double value);
  void setFloat(String path, float value);
  void setInt(String path, int value);
  void setLong(String path, long value);
  void setShort(String path, short value);
  void setBytes(String path, byte[] value);
  void setBigDecimal(String path, BigDecimal value);
  void setBigInteger(String path, BigInteger value);
  void setDataObject(String path, DataObject value);
  void setDate(String path, Date value);
  void setString(String path, String value);
  void setList(String path, List value);

  Object get(int propertyIndex);
  void set(int propertyIndex, Object value);
  boolean isSet(int propertyIndex);
  void unset(int propertyIndex);

  boolean getBoolean(int propertyIndex);
  byte getByte(int propertyIndex);
  char getChar(int propertyIndex);
  double getDouble(int propertyIndex);
  float getFloat(int propertyIndex);
  int getInt(int propertyIndex);
  long getLong(int propertyIndex);
  short getShort(int propertyIndex);
  byte[] getBytes(int propertyIndex);
  BigDecimal getBigDecimal(int propertyIndex);
  BigInteger getBigInteger(int propertyIndex);
  DataObject getDataObject(int propertyIndex);
  Date getDate(int propertyIndex);
  String getString(int propertyIndex);
  List getList(int propertyIndex);
  Sequence getSequence(int propertyIndex);

  void setBoolean(int propertyIndex, boolean value);
  void setByte(int propertyIndex, byte value);
  void setChar(int propertyIndex, char value);
  void setDouble(int propertyIndex, double value);
  void setFloat(int propertyIndex, float value);
  void setInt(int propertyIndex, int value);
  void setLong(int propertyIndex, long value);
  void setShort(int propertyIndex, short value);
  void setBytes(int propertyIndex, byte[] value);
  void setBigDecimal(int propertyIndex, BigDecimal value);
  void setBigInteger(int propertyIndex, BigInteger value);
  void setDataObject(int propertyIndex, DataObject value);
  void setDate(int propertyIndex, Date value);
  void setString(int propertyIndex, String value);
  void setList(int propertyIndex, List value);

  Object get(Property property);
  void set(Property property, Object value);
  boolean isSet(Property property);
  void unset(Property property);

  boolean getBoolean(Property property);
  byte getByte(Property property);
  char getChar(Property property);
  double getDouble(Property property);
  float getFloat(Property property);
  int getInt(Property property);
  long getLong(Property property);
  short getShort(Property property);
  byte[] getBytes(Property property);
  BigDecimal getBigDecimal(Property property);
  BigInteger getBigInteger(Property property);
  DataObject getDataObject(Property property);
  Date getDate(Property property);
  String getString(Property property);
  List getList(Property property);
  Sequence getSequence(Property property);

  void setBoolean(Property property, boolean value);
  void setByte(Property property, byte value);
  void setChar(Property property, char value);
  void setDouble(Property property, double value);
  void setFloat(Property property, float value);
  void setInt(Property property, int value);
  void setLong(Property property, long value);
  void setShort(Property property, short value);
  void setBytes(Property property, byte[] value);
  void setBigDecimal(Property property, BigDecimal value);
  void setBigInteger(Property property, BigInteger value);
  void setDataObject(Property property, DataObject value);
  void setDate(Property property, Date value);
  void setString(Property property, String value);
  void setList(Property property, List value);

  DataObject createDataObject(String propertyName);
  DataObject createDataObject(int propertyIndex);
  DataObject createDataObject(Property property);
  DataObject createDataObject(String propertyName, String namespaceURI,
String typeName);
```

```
  DataObject createDataObject(int propertyIndex, String namespaceURI,
String typeName);
  DataObject createDataObject(Property property, Type type);

  void delete();

  DataObject getContainer();
  Property getContainmentProperty();

  DataGraph getDataGraph();

  Type getType();
}
```

Sequence

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>, "annotation text", <`letters`, "A">, <`numbers`, 2>, <`letters`, "B">}

On the other hand, if a `DataObject` has the same two properties and is initialized in the same order (ignoring the unstructured text, which cannot be added to a `DataObject`), 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 preserved.

A `Sequence` is an ordered collection of *settings*, where each setting is either

- a combination of a property and a value, or
- a `String` containing unstructured text.

Note that the way in which a `DataObject` keeps track of the order of properties and values is quite different from the way this is done by a `Sequence:`

- In a `DataObject` the order of the properties is fixed (which means that each property has a well defined index) while 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.) A DataObject may not have any unstructured text.

- In a Sequence, the order of the settings across properties **is** preserved. A Sequence may have unstructured text interspersed among the settings.
- The same properties that appear in a sequence are also available through DataObject but without preserving the order across properties.
- The boolean add() accessors add to the end of the sequence. The add(int index) accessors add to the specified position in a sequence and, like java.util.List, shift entries at later positions upwards.
- The remove() accessor removes the entry at the specified index and shifts all later positions down.
- The move() accessor moves the entry at the fromIndex to the toIndex, shifting entries later than fromIndex down and after toIndex up.
- The setValue() accessor maintains positions and updates the value to be returned by getValue().

The ability to create DataObjects in a Sequence is part of the SDO roadmap.

```
public interface Sequence
{
  int size();

  Property getProperty(int index);
  Object getValue(int index);

  Object setValue(int index, Object value);

  boolean add(String propertyName, Object value);
  boolean add(int propertyIndex, Object value);
  boolean add(Property property, Object value);
  void add(int index, String propertyName, Object value);
  void add(int index, int propertyIndex, Object value);
  void add(int index, Property property, Object value);

  void remove(int index);
  void move(int toIndex, int fromIndex);
}
```

DataGraph

A DataGraph is a graph of DataObjects. The graph consists of a single root DataObject along with all the DataObjects that can be reached by recursively traversing the containment references of the root DataObject. A DataGraph forms a tree of DataObjects, where the non-containment references point to DataObjects within the tree. This is called *closure*.

```
public interface DataGraph
{
  DataObject getRootObject();

  DataObject createRootObject(String namespaceURI, String typeName);
  DataObject createRootObject(Type type);
  ChangeSummary getChangeSummary();

  Type getType(String uri, String typeName);
}
```

A DataGraph is created by a DMS, which returns either an empty DataGraph, or a DataGraph filled with DataObjects. The DMS is also responsible for creation of the metadata (i.e., model) used by the DataObjects and DataGraph. For example, a DMS for XML data could construct the model from the XSD for the XML.

A DataGraph may not have closure temporarily while the contained DataObjects are being modified by an end user through the DataObject interface, but after all user operations are completed the DataGraph is expected to be restored to closure. A DMS should operate only on DataGraphs with closure.

The DataGraph also 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 DataGraph is returned from a DMS. The client of the DMS makes modifications, which change the state of the DataObjects, including creation and deletion, with all a summary of changes recorded in the ChangeSummary.

The client may send a modified DataGraph to a DMS (the same or different depending on the scenario), at which time the DMS checks the DataGraph for errors. These errors include lack of closure of the DataGraph, values outside the lower and upper bounds of a property, choices spanning several properties or DataObjects, deferred constraints, or any restrictions specific to the DMS (e.g., XML Schema specific validations). The DMS will typically report problems with updates by throwing exceptions.

DataGraphs may be serialized to XML (see DataGraph Serialization), typically by an XML DMS.

The root DataObject is accessible by getRootObject(). An empty DataGraph may have a root assigned by the createRootObject() methods.

A Type may be accessed via getType(String uri, String typeName). 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, which may be a local copy or a resource on a network. The configuration information necessary to provide this logical to physical access to metadata is via implementation-specific configuration files. If the metadata is unavailable, an implementation-specific exception occurs.

The ability to set the root DataObject in a DataGraph is part of the SDO roadmap.

ChangeSummary

The `ChangeSummary` provides access to change history information for the `DataObjects` in a `DataGraph`. The change history covers any modifications that have been made to the `DataGraph` 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.

This interface includes methods that:

- Activate and deactivate logging and query the logging status.
- Access the `DataGraph` to which the `ChangeSummary` belongs.
- Access the changed `DataObjects`.
- Indicate whether an object has been created or deleted.
- Gets the list of the settings for the old values of any properties that have been modified.

The old values are expressed as a list of `ChangeSummary.Setting` objects. (Where `ChangeSummary.Setting` is an inner interface of `ChangeSummary`.) Each `ChangeSummary.Setting` has a value and a property, along with a flag to indicate whether or not the property is set.

Note that for a created object, the old values list is empty. For a deleted object, this list contains all the properties of the `DataObject`. For a `DataObject` that was modified, the list consists of only the modified properties.

```
public interface ChangeSummary
{
  void beginLogging();
  void endLogging();
  boolean isLogging();

  DataGraph getDataGraph();

  List /*DataObject*/ getChangedDataObjects();
  boolean isCreated(DataObject dataObject);
  boolean isDeleted(DataObject dataObject);

  public interface Setting
  {
    Property getProperty();
    Object getValue();
    boolean isSet();
  }

  List /*ChangeSummary.Setting*/ getOldValues(DataObject dataObject);
}
```

Type

The concept of a data type is common across most programming languages and data modeling languages. The `Type` interface represents a common view of a data type. A

`Type` has a set of `Property` objects. For example, in Java, C++, and EMOF, a Class represents a `Type` and each of their fields is represented by a `Property`. In XML Schema, SimpleType and ComplexType are represented by `Type`, with elements and attributes representing Properties. Type is also a concept in C as a Struct, where each field is a `Property`. Relational database tables are a similar analogy, where Table corresponds to `Type` and Column corresponds to `Property`. All of these domains share certain concepts, a small subset of which are represented here in the `Type` and `Property` interfaces. These interfaces are useful for `DataObject` programmers that need to introspect the shape or nature of data at runtime. More complete metamodel APIs (e.g., XML Schema or EMOF) representing all the information of a particular domain are outside the scope of this specification.

A `Type` has:

- *Name* – a `String` that is typically unique among the Types that belong to the same URI.
- *Uri* –The logical URI of a package or a target namespace, depending on your perspective.
- *Instance Class* – the `java.lang.Class` used to implement the SDO Type. Examples are `java.lang.Integer` and `DataObject`.
- *Properties* – a list of `Property` objects defined by this `Type`. Types corresponding to simple data types define no properties.

```
public interface Type
{
  String getName();
  String getURI();

  Class getInstanceClass();
  boolean isInstance(Object object);

  List /*Property*/ getProperties();
  Property getProperty(String propertyName);
}
```

Property

Each `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 a `DataObject` is sometimes called a *reference*; otherwise it is called an *attribute*.
- Each property can be either *single-valued* or *many-valued*.
- In the case of a reference, the property may be either a *containment* or *non-containment* reference. In EMOF the term containment reference is called composite. Containment properties are the parent-child relationships in a tree of DataObjects.
- A default value.
- A numeric index within the property's `Type`.

Each `Type` assigns a unique *index* to each property that belongs to the `DataObject` that is the same as the index in List returned by Type.getProperties().

Properties may have additional behavior specified by metadata. Metadata such as read-only, bi-directional opposite properties, resolving of hrefs and xlinks, and whether the property uses settings or is a sequence will affect the behavior of the `DataObject` accessors.

```
public interface Property
{
  String getName();
  Type getType();

  boolean isMany();
  boolean isContainment();

  Type getContainingType();

  Object getDefault();
}
```

DataGraph XML Serialization

A `DataGraph` may be serialized as an XML stream. If the metadata comes from XML Schema, the DataObjects are serialized following the XSD. If the metadata comes from another source, an XSD is generated and the DataObjects are serialized following the XSD using XMI [4] schema and document production rules, respectively.

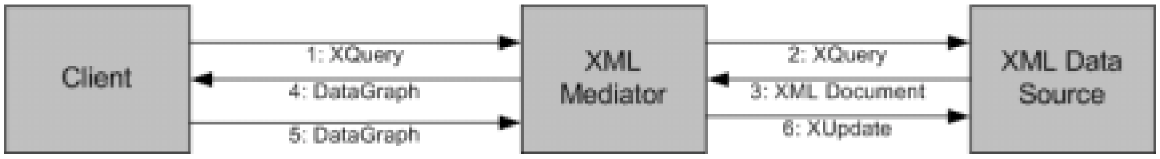

Figure 4: DataGraph XML Serialization

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 `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 DMS or `java.io.Serializable` or in a Web service, is expected to be the same XML format described here. When 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:complexType>

</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] extended with an additional ability to access data using the 0 as a base index, a style common throughout Java programming. Arrays and List.get(index) in Java both index from 0, and the intent is to enable the most productive environment for the Java programmer, avoiding the need for adding or subtracting 1 when using path expressions and Java indexes together. The syntax for specifying these paths, is shown here:

```
path ::= '/'? (step '/')* step
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
;; Only the last step have an attribute as the property
```

For example, consider the Company model described in Complete DataGraph for Company Example. 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:

```
    DataObject 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 "0002" is:

```
    DataObject employee =
        department.getDataObject("employees[SN='0002']");
```

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

```
    DataObject employee =
        company.getDataObject("departments.0/employees[SN='0002']");
```

If more than one property shares the same name, only the first is matched by the path expression, using property.getName() for name matching. 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.

Examples

The examples given here assume the use of an XML Data Mediator Service (`XMLDMS`) to load and save a `DataGraph` from and to XML files. The `XMLDMS` 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 `DataGraph` that was provided by any other `DMS`.

The examples covered here include:

1. Accessing DataObjects using XPath
2. Accessing DataObjects via Property Index
3. Accessing the Contents of a Sequence
4. Serializing/Deserializing a DataGraph or DataObject
5. Using Type and Property with DataObjects
6. Web services and DataGraphs Example

The example model is a Company with a Department containing several Employees. The XSD for the Company is shown in the Appendix, Complete DataGraph for Company Example.

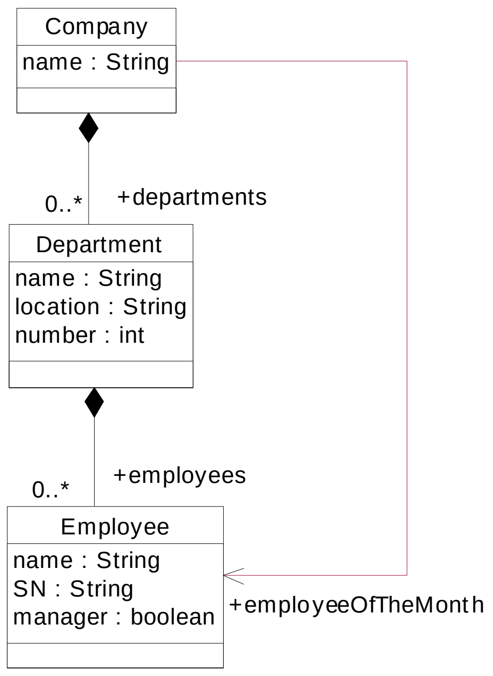

Figure 5: Data Model for Company

Accessing DataObjects using XPath

Assume that we have a `DataGraph` that was initialized by the `XMLDMS` from the following XML:

```
<sdo:datagraph xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
               xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">
  <company:company name="ACME" employeeOfTheMonth="#id.0">
    <departments name="Advanced Technologies" location="NY"
number="123">
      <employees name="John Jones" SN="0001"/>
      <employees xmi:id="id.0" name="Mary Smith" SN="0002"
manager="true"/>
      <employees name="Jane Doe" SN="0003"/>
    </departments>
  </company:company>
</sdo:datagraph>
```

(This XML conforms to the company model defined in Complete DataGraph for Company Example.)

In order to access the company `DataObject` from the root of the `DataGraph`, we could use the following code:

```
   // Access the company DataObject from the "company" property of
   // the root object.
   DataObject rootObject = dataGraph.getRootObject();
   DataObject 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.setString("name", " MegaCorp");
```

Now, suppose we wish to access the employee whose serial number is “0002”. If we know that this employee is located at index 1 within the department that is located at index 0 within the root company object, one way to do this is by traversing each reference in the `DataGraph` and locating each `DataObject` in a many-valued 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
    List departments = company.getList("departments");
    // Get the department at index 0 on the list
    DataObject department = (DataObject) departments.get(0);
    // Get the list of employees for the department
    List employees = department.getList("employees");
    // Get the employee at index 1 on the list
    DataObject employeeFromList = (DataObject) employees.get(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:
    DataObject 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
    DataObject employeeFromXPathByValue = company.getDataObject(
       "departments[number=123]/employees[SN='0002']");
```

In order to remove that employee from the `DataGraph`, we could use:

```
  // remove the employee from the graph
  employeeFromList.delete();
```

And, finally, to create a new employee:

```
  // create a new employee
  DataObject newEmployee =
               department.createDataObject("employees");
  newEmployee.set("name", "Al Smith");
  newEmployee.set("SN", "0004");
  newEmployee.setBoolean("manager", true);

  // Reset employeeOfTheMonth to be the new employee
  company.set("employeeOfTheMonth", newEmployee);
```

After saving this `DataGraph` using the `XMLDMS`, the resulting XML file would contain:

```
<sdo:datagraph xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
               xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">

  <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="0002"
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="0001"/>
      <employees xmi:id="id.3" name="Jane Doe" SN="0003"/>
      <employees xmi:id="id.4" name="Al Smith" SN="0004"
manager="true"/>
    </departments>
  </company:company>

</sdo:datagraph>
```

The `ChangeSummary` provides an overview of the changes that have been made to the `DataGraph`. The `ChangeSummary` contains `DataObject`s 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 `DataObject`s in the `ChangeSummary` back to the corresponding `DataObject`s in the `DataGraph`.

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 `DataGraph` are modified, the `ChangeSummary` would include only small subset of the overall `DataGraph`.

Note: The serialized `DataGraph` 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 `DataGraph` is shown in Complete DataGraph for Company Example.

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 indexes 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 indexes
 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;

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

 // Set the "name" property for the company
 company.setString(COMPANY_NAME, "MegaCorp");

 // Get the list of departments
 List departments = company.getList(COMPANY_DEPARTMENT);
 // Get the department at index 0 on the list
 DataObject department = (DataObject) departments.get(0);
 // Get the list of employees for the department
 List employees = department.getList(DEPARTMENT_EMPLOYEES);
 // Get the employee at index 1 on the list
 DataObject employeeFromList = (DataObject) employees.get(1);

 // remove the employee from the graph
 employeeFromList.delete();

 // create a new employee
 DataObject newEmployee =
               department.createDataObject(DEPARTMENT_EMPLOYEES);
 newEmployee.set(EMPLOYEE_NAME, "Al Smith");
 newEmployee.set(EMPLOYEE_SN, "0004");
 newEmployee.setBoolean(EMPLOYEE_MANAGER, true);
```

Accessing the Contents of a Sequence

The following code uses the `Sequence` interface to analyze the contents of a `DataGraph` 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:

```
public static void checkSequence(DataGraph dataGraph)
{
 // Access the FormLetter DataObject from the "letters"
 // property of the root object.
 DataObject rootObject = dataGraph.getRootObject();
 DataObject letterDataObject = rootObject.getDataObject("letters");
```

```
  // Access the mixed text Sequence of the FormLetter
  Sequence letterSequence = letterDataObject.getSequence("mixed");

  // Print out all the settings that contain unstructured text
  System.out.println("Unstructured text:");
  for (int i=0; i<letterSequence.size(); i++)
  {
    String propertyName = letterSequence.getPropertyName(i);
    if (propertyName==null)
    {
      String text = (String) letterSequence.getValue(i);
      System.out.println("\t("+text+")");
    }
  }

  // Verify that the lastName property of the DataObject has the same
  // value as the lastName property for the Sequence.
  String dataObjectLastName = letterDataObject.getString("lastName");
  for (int i=0; i<letterSequence.size(); i++)
  {
    String propertyName = letterSequence.getPropertyName(i);
    if ("lastName".equals(propertyName))
    {
      String sequenceLastName = (String)letterSequence.getValue(i);
      if (dataObjectLastName == sequenceLastName)
        System.out.println("Last Name property matches");
      break;
    }
  }
}
```

Assume that the following XML file is loaded by the `XMLDMS` to produce a `DataGraph` that is passed to the `checkSequence(DataGraph)` method:

```
<sdo:datagraph xmlns:sdo="commonj.sdo"
            xmlns:letter="http://letterSchema">
<!-- Letter Schema -->
  <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>
```

(Note: this XML conforms to the schema defined in XSD Schema for Letter Model.)

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
```

Serializing/Deserializing a DataGraph or DataObject

The `DataObject` and `DataGraph` interfaces extend `java.io.Serializable`, so any `DataObject` and `DataGraph` can be serialized. For example, the following code can be used to serialize a given `DataObject` into a file with a given name:

```
public void serializeDO(DataObject dataObject, String fileName)
{
  try
  {
     // serialize data object
     FileOutputStream fos = new FileOutputStream(fileName);
     ObjectOutputStream out = new ObjectOutputStream(fos);
     out.writeObject(dataObject);
     out.flush();
     out.close();
     fos.close();
  }
     catch (IOException e)
  {
     e.printStackTrace();
     throw new WrappedException(e);
  }
}
```

The following code can be used to deserialize a DataObject from a file with a given name:

```
public DataObject deserializeDO(String fileName)
{
  try
  {
     //  de-serialize
     FileInputStream fis = new FileInputStream(fileName);
     ObjectInputStream input = new ObjectInputStream(fis);
     DataObject dataObject = (DataObject) input.readObject();
     input.close();
     fis.close();
     return dataObject;
   }
    catch (IOException e)
  {
     e.printStackTrace();
     throw new WrappedException(e);
  }
     catch (ClassNotFoundException e)
  {
      e.printStackTrace();
      throw new WrappedException(e);
    }
  }
```

Similarly, the following code can be used to serialize and deserialize a `DataGraph`:

```
  public void serializeDG(DataGraph dataGraph, String fileName)
  {
    try
    {
      // serialize data graph
      FileOutputStream fos = new FileOutputStream(fileName);
      ObjectOutputStream out = new ObjectOutputStream(fos);
      out.writeObject(dataGraph);
      out.flush();
      out.close();
      fos.close();
    }
      catch (IOException e)
    {
      e.printStackTrace();
      throw new WrappedException(e);
    }
  }
  public DataGraph deserializeDG(String fileName)
  {
    try
    {
      //  de-serialize
      FileInputStream fis = new FileInputStream(fileName);
      ObjectInputStream input = new ObjectInputStream(fis);
      DataGraph deserializedDataGraph = (DataGraph) input.readObject();
      input.close();
      fis.close();
      return deserializedDataGraph;
    }
      catch (IOException e)
    {
      e.printStackTrace();
      throw new WrappedException(e);
    }
      catch (ClassNotFoundException e)
    {
      e.printStackTrace();
      throw new WrappedException(e);
    }
  }
```

Using Type and Property with DataObjects

The Type interface provides access to the metadata for DataObjects in a DataGraph. The methods on Type and Property provide information that describes the properties a DataObject in the DataGraph. 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.

```
public void printDataObject(DataObject dataObject, int indent)
{
  // Retrieve the Type
  Type type = dataObject.getType();

  // For each Property
  List properties = type.getProperties();
  for (int p=0, size=properties.size(); p < size; p++)
  {
    if (dataObject.isSet(p))
    {
      Property property = (Property) properties.get(p);
      if (property.isMany())
      {
        // For many-valued properties, process a list of values
        List values = dataObject.getList(p);
        for (int v=0, count=values.size(); v < count; v++)
        {
          printValue(values.get(v), property, indent);
        }
      }
      else
      {
        // For single-valued properties, print out the value
        printValue(dataObject.get(p), property, indent);
      }
    }
  }
}

void printValue(Object value, Property property, int indent)
{
  // Get the name of the property
  String propertyName = property.getName();

  // Construct a string for the proper indentation
  String margin = "";
  for (int i = 0; i < indent; i++)
    margin += "\t";

  if (value != null && property.isContainment())
  {
    // For containment properties, display the value with
printDataObject
    String typeName = property.getType().getName();
    System.out.println( margin + propertyName + " (" +typeName+ "):");
```

```
      printDataObject((DataObject) value, indent + 1);
    }
    else
    {
      // For non-containment properties, just print the value
      System.out.println(margin + propertyName + ": " + value);
    }
  }
```

For example, consider the following XML file:

```
<sdo:datagraph xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
               xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">
  <company:company name="ACME" employeeOfTheMonth="#id.0">
    <departments name="Advanced Technologies" location="NY"
number="123">
      <employees name="John Jones" SN="0001"/>
      <employees xmi:id="id.0" name="Mary Smith" SN="0002"
manager="true"/>
      <employees name="Jane Doe" SN="0003"/>
    </departments>
  </company:company>
</sdo:datagraph>
```

(Note: this XML conforms to the company model XSD defined in Complete DataGraph for Company Example.)

If this file is loaded using an XML Data Mediator Service, the resulting `DataGraph` could be printed out using:

```
   printDataObject(dataGraph.getRootObject(), 0);
```

The console output for this `DataGraph` would be:

```
company (Company):
  name: ACME
  departments (Department):
    name: Advanced Technologies
    location: NY
    number: 123
    employees (Employee):
      name: John Jones
      SN: 0001
    employees (Employee):
      name: Mary Smith
      SN: 0002
      manager: true
    employees (Employee):
      name: Jane Doe
      SN: 0003
  employeeOfTheMonth: Employee (name=Mary Smith, SN=0002,
               manager=true, employeeStatus=fullTime)
```

Web services and DataGraphs Example

DataGraphs 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 xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
      xmlns:company="company.xsd"
      xmlns:sdo="commonj.sdo">
      <company:company name="ACME" employeeOfTheMonth="#id.0">
        <departments name="Advanced Technologies" location="NY"
number="123">
          <employees name="John Jones" SN="0001"/>
          <employees xmi:id="id.0" name="Mary Smith" SN="0002"
manager="true"/>
          <employees name="Jane Doe" SN="0003"/>
        </departments>
      </company:company>
    </sdo:datagraph>
  </soap:Body>
</soap:Envelope>
```

The SDO `DataGraphType` 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 DataGraph for Company Example.

```
<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:anyAttribute namespace="http://www.omg.org/XMI"
processContents="lax"/>
</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 DataGraph. 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 xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
      xmlns:company="company.xsd">
      <company:company name="ACME" employeeOfTheMonth="#id.0">
        <departments name="Advanced Technologies" location="NY"
number="123">
          <employees name="John Jones" SN="0001"/>
          <employees xmi:id="id.0" name="Mary Smith" SN="0002"
manager="true"/>
          <employees name="Jane Doe" SN="0003"/>
        </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.

```
<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>
```

Appendix 1: DataGraph Example

Complete DataGraph Serialization

As mentioned in the section on DataGraph Serialization, the serialization of the `DataGraph` includes optional elements that describe the model the change information in addition to the `DataObject`s in the `DataGraph`.

The model may be described either as an instance of an XML Schema or EMOF Package (See Complete DataGraph for Company Example) or using an XML Schema (see Complete DataGraph for Letter Example.)

Complete DataGraph for Company Example

The following XML represents the complete serialization of the `DataGraph` that includes the changes from the processing described in Accessing `DataObject`s using XPaths.

```
<sdo:datagraph xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
               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:anyAttribute namespace="http://www.omg.org/XMI"
processContents="lax"/>
      </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:anyAttribute namespace="http://www.omg.org/XMI"
processContents="lax"/>
      </xsd:complexType>
      <xsd:complexType name="EmployeeType">
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="SN" type="xsd:string"/>
        <xsd:attribute name="manager" type="xsd:boolean"/>
        <xsd:anyAttribute namespace="http://www.omg.org/XMI"
processContents="lax"/>
      </xsd:complexType>
    </xsd:schema>
  </xsd>
  <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="0002"
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="0001"/>
      <employees xmi:id="id.3" name="Jane Doe" SN="0003"/>
      <employees xmi:id="id.4" name="Al Smith" SN="0004"
manager="true"/>
    </departments>
  </company:company>
</sdo:datagraph>
```

When using EMOF as metadata, the complete DataGraph 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="0002"
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="0001"/>
      <employees xmi:id="id.3" name="Jane Doe" SN="0003"/>
      <employees xmi:id="id.4" name="Al Smith" SN="0004"
manager="true"/>
    </departments>
  </company:company>
</sdo:datagraph>
```

Complete DataGraph for Letter Example

This `DataGraph` is used as the input for the example shown in Accessing the Contents of a Sequence. 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 `DataGraph`, the XSD is the metadata and the letter is the data.

```
<sdo:datagraph 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>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 DataGraph 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>
```

Appendix 2: Type Conversions

The type conversions marked X and x are supported, based on the Java 1.4 APIs [6]. X is an identity, and x is a call to a conversion API such as Integer.longValue(). Conversions between the primitive and object wrapper form are also supported.

To-> From | V	boolean	byte	char	double	float	int	long	short	String	byte[]	BigDecimal	BigInteger	Date
boolean	X								x				
byte		X		x	x	x	x	x	x				
char			X						x				
double		x		X	x	x	x	x	x		x	x	
float		x		x	X	x	x	x	x		x	x	
int		x		x	x	X	x	x	x		x	x	
long		x		x	x	x	X	x	x		x	x	x
short		x		x	x	x	x	X	x				
String	x	x	x	x	x	x	x	x	X		x	x	
byte[]										X		x	
BigDecimal				x	x	x	x		x		X	x	
BigInteger				x	x	x	x		x	x	x	X	
Date							x						X

Acknowledgements

We would like to thank Johsua Auerbach (IBM), David Bau (BEA), Adam Bosworth (BEA), Graham Barber (IBM), Kevin Bauer (IBM), Michael Beisiegel (IBM), Frank Budinsky (IBM), Shane Claussen (IBM), Ed Cobb (BEA), Brent Daniel (IBM), George DeCandio (IBM), Jean-Sebastien Delfino (IBM), Scott Dietzen (BEA), Mike Edwards

(IBM), Don Ferguson (IBM), Christopher Ferris (IBM), Paul Fremantle (IBM), Kelvin Goodson (IBM), Laurent Hasson (IBM), Rob High (IBM), Steve Holbrook (IBM), Sridhar Iyengar (IBM), Jagan Karuturi (IBM), Matthew Lovett (IBM), Angel Luis Diaz (IBM), Ed Merks (IBM), Adam Messinger (BEA), Peter Niblett (IBM), Karla Norsworthy (IBM), Barbara Price (IBM), Fabio Riccardi (BEA), Timo Salo (IBM), Greg Truty (IBM), Celia Tung (IBM), Seth White (BEA), Kevin Williams (IBM), and George Zagelow (IBM).

Trademarks

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, http://dev2dev.bea.com/technologies/commonj/index.jsp
or at this location:
http://www.ibm.com/developerworks/library/j-commonj-sdowmt/

[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 1.4.1 API documentation http://java.sun.com/j2se/1.4.2/docs/index.html