If you think the J2EE programming models and APIs force developers to spend too much time on technology-specific configuration, programming, and debugging, then this article is for you! Many Java™ developers are skeptical about how heterogeneous data can be accessed uniformly, and have been disappointed in the various programming frameworks that propose to solve the problem. In this article, Java developers Bertrand Portier and Frank Budinsky introduce you to next-generation data programming with Service Data Objects (SDO).

Put simply, SDO is a framework for data application development, which includes an architecture and API. SDO does the following:

- Simplifies the J2EE data programming model
- Abstracts data in a service oriented architecture (SOA)
- Unifies data application development
- Supports and integrates XML
- Incorporates J2EE patterns and best practices

In this introduction to the SDO framework, we will try to explain the motivation behind the SDO effort and the differences between SDO and other specifications. Then, we will describe the components that make SDO. Finally, you will have a chance to see SDO in action as we describe a sample SDO application.

Why SDO?

The first question most developers will ask about Service Data Objects (SDO) is why. Isn't J2EE big and complex enough (and hard enough to learn) as it is? Also, other frameworks already support XML in the Java environment, don't they? The answer, fortunately, is one that should make most of us quite happy: SDO emerged as a means of simplifying the J2EE data programming model, thus giving J2EE developers more time to focus on the business logic of their applications.

The Service Data Objects framework provides a unified framework for data application development. With SDO, you do not need to be familiar with a technology-specific API in order to
access and utilize data. You need to know only one API, the SDO API, which lets you work with data from multiple data sources, including relational databases, entity EJB components, XML pages, Web services, the Java Connector Architecture, JavaServer Pages pages, and more.

Note that we used the word *framework*. This is analogous to the Eclipse framework. Eclipse is designed so that tools can be integrated together thanks to its solid and extensible base. SDO is similar in the sense that it provides a framework to which applications can be contributed and these applications will all be consistent with the SDO model.

Unlike some of the other data integration models, SDO doesn't stop at data abstraction. The SDO framework also incorporates a good number of J2EE patterns and best practices, making it easy to incorporate proven architecture and designs into your applications. For example, the majority of Web applications today are not (and cannot) be connected to backend systems 100 percent of the time; so SDO supports a disconnected programming model. Likewise, today's applications tend to be remarkably complex, comprising many layers of concern. How will data be stored? Sent? Presented to end users in a GUI framework? The SDO programming model prescribes patterns of usage that allow clean separation of each of these concerns.

XML is becoming ubiquitous in distributed applications. For example, XML Schema (XSD) is used to define business rules in an application's data format. Also, XML itself is used to facilitate interaction: Web services use XML-based SOAP as the messaging technology. XML is a very important driver of SDO and is supported and integrated in the framework.

Comparing technologies

As we previously mentioned, SDO isn't the only technology that proposes to resolve the problem of data integration in distributed applications. In this section, we'll see how SDO stacks up against similar programming frameworks such as JDO, JAXB, and EMF.

**SDO and WDO**

Web Data Objects, or WDO, is the name of an early release of SDO shipped in IBM WebSphere® Application Server 5.1 and IBM WebSphere Studio Application Developer 5.1.2. If you've spent any time with WebSphere Studio 5.1.2, you should already be somewhat familiar with SDO, although you're probably accustomed to seeing it denoted as WDO, for example in library names. Forget WDO, it's called SDO now!

**SDO and JDO**

JDO stands for Java Data Objects. JDO has been standardized through the Java Community Process (JCP) with 1.0 and maintenance release 1.0.1 in May 2003. A JCP expert group is being formed for version 2.0. JDO looks at data programming in the Java environment and provides a common API to access data stored in various types of data sources; for example, databases, file systems, or transaction processing systems. JDO preserves relationships between Java objects (graphs) and at the same time allows concurrent access to the data.

JDO's goal is similar to SDO's in that it wants to simplify and unify Java data programming so that developers can focus on business logic instead of the underlying technology. The main
difference, however, is that JDO looks at the persistence issue only (the J2EE data tier or enterprise information system (EIS) tier), whereas SDO is more general and represents data that can flow between any J2EE tier, such as between a presentation and business tier.

Interestingly, SDO can be used in conjunction with JDO where JDO is a data source that SDO can access, applying the Data Transfer Object (DTO) design pattern. Similarly, SDO can be used in conjunction with entity EJB components and the Java Connector Architecture (JCA), the intent being to provide uniform data access.

**SDO and EMF**

EMF stands for Eclipse Modeling Framework. Based on a data model defined using Java interfaces, XML Schema, or UML class diagrams, EMF will generate a unifying metamodel (called Ecore) which in conjunction with the framework can be used to create a high-quality implementation of the model. EMF provides persistence, a very efficient reflective generic object manipulation API, and a change-notification framework. EMF also includes generic reusable classes for building EMF model editors.

EMF and SDO both deal with data representation. In fact, IBM's reference implementation of SDO, which we'll use later in this article, is an EMF implementation of SDO. EMF code generation was even used to create some of the SDO implementation, based on a UML model definition of SDO itself. The implementation of SDO is essentially a thin layer (facade) over EMF and is packaged and shipped as part of the EMF project. See [Related topics](#) for more information on EMF.

**SDO and JAXB**

JAXB stands for Java API for XML Data Binding. JAXB 1.0 was released by the JCP in January 2003. The JCP expert group has produced an early draft for version 2.0. JAXB is about XML data binding; that is, representing XML data as Java objects in memory. As an XML binding framework for the Java language, JAXB saves you from having to parse or create XML documents yourself. (In fact, it saves you from having to deal with the XML at all.) JAXB performs the marshalling/serializing (Java to XML) and unmarshalling/deserializing (XML to Java) for you.

SDO defines a Java binding framework of its own, but it goes one step further. While JAXB is only focused on a Java-to-XML binding, XML isn't the only kind of data being bound to SDO. As stated previously SDO provides uniform access to data of various types, only one of which is XML. SDO also offers both a static and dynamic API*, whereas JAXB only provides a static binding.

*Note that the sample application for this article utilizes only dynamic SDO, although the EMF code generator also provides full support for static code generation of data objects.

**SDO and ADO .NET**

ADO used to stand for ActiveX Data Objects, but it is no longer true in the .NET context. ADO .NET provides uniform data access between different tiers in the .NET framework.

ADO .NET and SDO share similar motivators for supporting XML and applications distributed on multiple tiers. Other than technical differences, one major difference between the two technologies
is that ADO .NET is for the Microsoft .NET platform and is a proprietary technology, whereas SDO is for the Java (J2EE) platform and is being standardized through the Java Community Process.

**SDO components**

In this section, we'll provide an architectural overview of SDO. We'll describe each of the components that make up the framework and explain how they work together. The first three components we'll discuss are "conceptual" features of SDO: They do not have a corresponding interface in the API.

**SDO clients**

SDO clients use the SDO framework to work with data. Instead of using technology-specific APIs and frameworks, they use the SDO programming model and API. SDO clients work on SDO data graphs (see Figure 1) and do not need to know how the data they are working with is persisted or serialized.

**Data mediator services**

Data mediators services (DMSs) are responsible for creating a data graph from data source(s), and updating data source(s) based on changes made to a data graph. A data mediator framework is not in the scope of the SDO 1.0 specification; in other words, SDO 1.0 doesn't talk about specific DMSs. Examples of DMSs include: a JDBC DMS, an entity EJB DMS, an XML DMS, etc.

**Data sources**

Data sources are not restricted to back-end data sources (for example, persistence databases). A data source contains data in its own format. Only DMSs access data sources, SDO applications do not. SDO applications may only work with data objects in data graphs.

Each of the following components corresponds to a Java interface in the SDO programming model. The SDO reference implementation (see Related topics) provides EMF-based implementations of these interfaces.

**Data objects**

Data objects are the fundamental components of SDO. In fact, they are the service data objects found in the name of the specification itself. Data objects are the SDO representation of structured data. Data objects are generic and provide a common view of structured data built by a DMS. While a JDBC DMS, for instance, needs to know about the persistence technology (for example, relational databases) and how to configure and access it, SDO clients need not know anything about it. Data objects hold their "data" in properties (more on properties in a moment). Data objects provide convenience creation and deletion methods (createDataObject() with various signatures and delete()) and reflective methods to get their types (instance class, name, properties, and namespaces). Data objects are linked together and contained in data graphs.

**Data graphs**

Data graphs provide a container for a tree of data objects. They are produced by the DMS for SDO clients to work with. Once modified, data graphs are passed back to the DMS for updating
the data source. SDO clients can traverse a data graph and read and modify its data objects. SDO is a disconnected architecture because SDO clients are disconnected from the DMS and the data source; they only see the data graph. Furthermore, a data graph can include objects representing data from different data sources. A data graph contains a root data object, all of the root's associated data objects, and a change summary (more on change summaries in a moment). When being transmitted between application components (for example, between a Web service requester and provider during service invocation), to the DMS, or saved to disk, data graphs are serialized to XML. The SDO specification provides the XML Schema of this serialization. Figure 1 shows an SDO data graph.

**Figure 1. An SDO data graph**

![Image of SDO data graph]

**Change summary**

Change summaries are contained by data graphs and are used to represent the changes that have been made to a data graph returned by the DMS. They are initially empty (when the data graph is returned to a client) and populated as the data graph is modified. Change summaries are used by the DMS at backend update time to apply the changes back to the data source. They allow DMSs to efficiently and incrementally update data sources by providing lists of the changed properties (along with their old values) and the created and deleted data objects in the data graph. Information is added to the change summary of a data graph only when the change summary's logging is activated. Change summaries provide methods for DMSs to turn logging on and off, as we'll describe in more detail in the sample application section.

**Properties, types, and sequences**

Data objects hold their contents in a series of properties. Each property has a type, which is either an attribute type such as a primitive (for example, `int`) or a commonly used data type (for example, `Date`) or, if a reference, the type of another data object. Each data object provides read and write access methods (getters and setters) for its properties. Several overloaded versions of these accessors are provided, allowing the properties to be accessed by passing the property name (`String`), number (`int`), or property meta object itself. The `String` accessor also supports an XPath-like syntax for accessing properties. For example you can call `get("department[number=123]")` on a company data object to get its first department whose number is 123. Sequences are more advanced. They allow order to be preserved across heterogeneous lists of property-value pairs.
**SDO and sample installation**

Enough concepts and theory! It's time for some hands-on practice. The good news is you can use SDO today, and for free! In this section, we provide an SDO sample application that runs on IBM's reference implementation of SDO, which is packaged as part of the Eclipse Modeling Framework (EMF). We'll first describe how to install EMF 2.0.1 (which includes SDO), and then show you how to set up the sample application provided with this article.

**Installing EMF 2.0.1**

If you already have EMF 2.0.1 installed, or if you know how to install it, skip to the next section.

IBM's implementation of SDO 1.0 is packaged with EMF 2.0.1. You need to install EMF 2.0.1* to use SDO. You can use the eclipse update manager method, described on the EMF site, or follow the steps below.

*An implementation of SDO 1.0 was also available in EMF 2.0.0.*

On the [EMF home page](http://www.eclipse.org/modeling/emf/), you'll find a collection of download links under the Quick Nav section. You want the "v2.x: EMF and SDO" download option. Make sure you read the installation requirements before you install EMF. Basically, you need to have Eclipse 3.0.1 and a Java Development Kit (JDK) 1.4 installed before you install EMF 2.0.1. Make sure you choose the EMF 2.0.1 release build. We recommend "All" as the type of package: emf-sdo-xsd-SDK-2.0.1.zip, so that you get source, runtime and docs all in one file. If you prefer, you can download the minimum package for SDO, which is labeled "EMF & SDO RT": emf-sdo-runtime-2.0.1.zip.

Extract the zip file to where eclipse was extracted (files in the archive are structured as eclipse/plug-ins/...). To check that the EMF installation was successful, launch Eclipse and then select Help> About the Eclipse Platform. Click the Plug-in Details button. Make sure the org.eclipse.emf.* plug-ins are at the 2.0.1 level. The following six plug-ins relate to SDO:

- org.eclipse.emf.commonj.sdo
- org.eclipse.emf.ecore.sdo
- org.eclipse.emf.ecore.sdo.doc
- org.eclipse.emf.ecore.sdo.edit
- org.eclipse.emf.ecore.sdo.editor
- org.eclipse.emf.ecore.sdo.source

Only the two plug-ins org.eclipse.emf.commonj.sdo and org.eclipse.emf.ecore.sdo are needed at runtime and they may be the only ones you see if you chose to install the runtime plug-ins only. That's it for the EMF installation.

**Installing the sample SDO application**

The next step is to add the SDO sample application for this article to your workspace. Follow these steps:

1. Launch Eclipse and create a new Plug-In Project.
2. Name the project **SDOSample** and create a Java source project with source folder `src` and output folder `bin`.

3. Click **Next**.

4. Deselect the "Generate the Java class that controls the plug-in's life cycle" option and click **Finish**.

Next, click on the **Code** icon at the top or bottom of this article (or see the **Download** section) to get the j-sdoSample.zip. Extract it to the SDOSample directory (From within the project in Eclipse: `Import... > Zip file`). Make sure you keep the folder structure and overwrite existing files. The SDOSample project is now populated with the files from j-sdoSample.zip.

**Note:** SDOSample is packaged as an Eclipse plug-in project so that you don’t have to set the library dependencies yourself. However, the sample is just Java code, so it could also be run as a standalone application as long as the CLASSPATH includes the EMF and SDO libraries (JAR files).

Your environment should now look something like the screenshot shown in Figure 2.

**Figure 2. Eclipse environment**

We're now ready to begin using our sample SDO application.

**A simple SDO application**

The example application we'll use for the remainder of the article is limited in terms of functionality, but it will help you understand SDO better. The application comes in two parts, which are separated into two corresponding packages: dms and client.

SDO 1.0 doesn't specify a standard DMS API. So, for this example we've designed our own DMS interface that provides two methods, as shown in Listing 1.
Listing 1. The DMS interface

```java
/**
 * A simple Data Mediator Service (DMS) that builds
 * SDO Data Graphs of Employees and updates
 * a backend data source according to a Data Graph.
 */
public interface EmployeeDMS {
    /**
     * @param employeeName the name of the employee.
     * @return an SDO Data Graph with Data Objects for
     * that employee's manager, that employee,
     * and that employee's "employees".
     */
    DataGraph get(String employeeName);

    /**
     * updates backend data source according to dataGraph.
     * @param dataGraph Data Graph used to update data source.
     */
    void update(DataGraph dataGraph);
}
```

The client instantiates a DMS and calls its `get()` method for specific employees: The Big Boss, Wayne Blanchard, and Terence Shorter. It prints information about these employees to the console in a user-friendly way, then updates department information for Terence Shorter and his employees. Finally, it calls the DMS's `update()` method, passing the updated data graph for Terence Shorter.

Note that for demonstration purposes we did not implement a data source component. Instead, the DMS has "hardcoded" knowledge of how to build the data graph based on the query. Figure 3 shows the employee hierarchy the DMS is using.

Figure 3. The employees of the Big Boss Corporation

As you can see, the virtual company behind the DMS has four employees. The company hierarchy is as follows:

- The Big Boss has no manager and Terence Shorter as his direct report.
- Terence Shorter has The Big Boss as his manager, and John Datrane and Miles Colvis as his direct reports.
- John Datrane has Terence Shorter as his manager and no direct reports.
• Miles Colvis has Terence Shorter as his manager and no direct reports.

**Running the example**

To run the example application, right-click SDOClient.java, then select Run>Java application. You should see something similar to Listing 2 on your console.

**Listing 2. The application's console output**

```
********** EMPLOYEE INFORMATION **********
Name: John Datrane
Number: 4
Title: Mr.
Department: Procurement
Is manager?: no

DIRECT MANAGER:
Name: Terence Shorter
Number: 2
Title: Mr.
Department: Financing
Is manager?: yes

************* EMPLOYEE INFORMATION *************
NO INFORMATION AVAILABLE ON EMPLOYEE Wayne Blanchard

********** EMPLOYEE INFORMATION **********
Name: Terence Shorter
Number: 2
Title: Mr.
Department: Financing
Is manager?: yes

DIRECT MANAGER:
Name: The Big Boss
Number: 1
Title: Mr.
Department: Board
Is manager?: yes

DIRECT EMPLOYEES:
Name: Miles Colvis
Number: 3
Title: Mr.
Department: Accounting
Is manager?: no

Name: John Datrane
Number: 4
Title: Mr.
Department: Procurement
Is manager?: no

[Total: 2]

DMS updating Terence Shorter
(changed department from "Financing" to "The new department")
DMS updating Miles Colvis
```
Now, let's see how each of the application's components works.

The client

The SDO client instantiates a DMS and gets data graphs for various employees from it. Once it gets a data graph, it navigates andaccesses data objects through the root object (using SDO's dynamic API), as shown here:

```java
// Get the SDO DataGraph from the DMS.
DataGraph employeeGraph = mediator.get(employeeName);
...
// Get the root object
DataObject root = employeeGraph.getRootObject();
...
// get the employee under the manager
employee = theManager.getDataObject("employees.0");
```

The client then calls the dynamic SDO accessor API to get information out of data objects and print it to the console, as shown here:

```java
System.out.println("Name: " + employee.getString("name"));
System.out.println("Number: " + employee.getInt("number");
...
System.out.println("Is manager?: " +
    (employee.getBoolean("manager") ? "yes" : "no") + "\n");
```

We've seen how the client gets information out (reading), but what about writing? More specifically, how does the client modify objects? To update data objects, SDO clients typically use `DataObject` write accessor methods. For example, here we can see how the client modifies the data graph obtained for the employee Terence Shorter:

```java
employee.setString("department", newDepartmentName);
```

Note the client doesn't call the logging methods. The DMS takes care of logging by calling `beginLogging()` and `endLogging()` on the data graph's change summary.

The data graph

The data format (model) of the data graph can be considered a contract between the DMS and the client. It is what the client expects from the DMS and what the DMS knows how to build (and also to read from to update back-end data sources). If you're familiar with XML or Web services, you can think of the data graph model as the XML Schema (XSD) that defines your data objects. The data graphs themselves would then be analogous to XML instance documents. As a matter of fact, XML Schema is one of the ways that an SDO model can be defined.

Note that data graphs and their models are always serializable to XML. In `SDOClient.java`, set the `debug` variable to `true` and you should see the serialized version of the result data graph on the console at runtime. It should look something like what you see in Listing 3.
Listing 3. The serialized version of the data graph

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sd0:datagraph xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:company="http://com.example.company.ecore"
xmlns:sdo="commonj.sdo"
xmlns:sdo_1="http://www.eclipse.org/emf/2003/SDO">
  <changeSummary>
    <objectChanges key="#//@eRootObject">
      <value xsi:type="sdo_1:EChangeSummarySetting" featureName="department" dataValue="Board"/>
    </objectChanges>
  </changeSummary>
  <company:Employee name="The Big Boss" number="1" department="The new department" title="Mr." manager="true">
    <employees name="Terence Shorter" number="2" department="The new department" title="Mr." manager="true">
      <employees name="Miles Colvis" number="3" department="The new department" title="Mr."/>
      <employees name="John Datrane" number="4" department="The new department" title="Mr."/>
    </employees>
  </company:Employee>
</sd0:datagraph>
```

For this example, the data graph is made of Employee data objects (and a change summary). An Employee has attributes such as name, number, department, title, manager (another employee who is the manager for that employee), and employees (other employees managed by that employee).

In this example, when the employee exists in the hardcoded data source, the data graph returned by the DMS will always be in the form of the employee's manager (if there is one), the employee requested, and his/her direct employees (if any).

The DMS: Building a graph

SDO 1.0 doesn't specify a DMS API, which would include the design and creation of the data graph model itself. Designing a data graph could be the subject of another article on its own as there are many scenarios to consider when building access to a data source.

For this example, we'll work with an employee model defined by the DMS using the dynamic EMF API. The example data graph has no model document such as XSD. The fact that data objects have been dynamically generated means no Employee Java classes have been generated. Had the static method been used, the opposite would be true.

DMSs get their information from various data sources using various data access APIs (JDBC, SQL, etc.). However, once the information is retrieved from the backend (this example simply has hard-coded knowledge), the DMS uses EMF APIs (eGet, eSet), instead of the SDO ones, to build the data graph of data objects. This approach yields optimal performance but has the disadvantage of not being portable across SDO implementations.

In cases where performance isn't a major concern, this same DMS design could be implemented using SDO APIs. In that case, the cached meta objects in the DMS class (employeeClass, employeeNameFeature, etc.) would be of types commonj.sdo.Type and commonj.sdo.Property, instead of the EMF types EClass, EAttribute, and EReference. Furthermore, if performance
is of no concern at all, the convenient String-based SDO APIs (such as setBoolean(String path, boolean value)) could be used, thus eliminating the need to cache the meta objects. Unfortunately, while more convenient, that solution would run much slower.

The code snippet below shows how the Employee model is defined, in SimpleEmployeeDataMediatorImpl.java. This isn't the code to build SDO objects yet; it's just the model of the SDO objects:

```java
protected EClass employeeClass;
protected EAttribute employeeNameFeature;
protected EReference employeeEmployeesFeature;
...
employeeClass = ecoreFactory.createEClass();
employeeClass.setName("Employee");
EAttribute employeeNameFeature = ecoreFactory.createEAttribute();
...
// employees (that the employee manages)
employeeEmployeesFeature = ecoreFactory.createEReference();
employeeEmployeesFeature.setContainment(true);
...
EPackage employeePackage = ecoreFactory.createEPackage();
employeePackage.getEClassifiers().add(employeeClass);
...
```

Note that we call setContainment with the value true on the employees EReference, so that each employee will "contain" his or her employees. If we didn't do this the nested employees would not be in (that is, contained by) the data graph and the change summary would not include modifications to employees other than the root object of the graph.

### Modeling SDO

At this point, you're probably thinking, "Interesting but this will give me EMF objects and not SDO data objects. What's the trick here?" Well, it's straightforward. The Employee EClass belongs to the employeePackage EPackage, which has the following call:

```java
// Have the factory for this package build SDO Objects
employeePackage.setEFactoryInstance(
  new DynamicEDataObjectImpl.FactoryImpl());
```

At runtime the factory will create objects of type DynamicEDataObjectImpl, which implements the DataObject interface (that is, SDO data objects), rather than the default DynamicEObjectImpl, which would create only ordinary EMF objects. This highlights the relationship between SDO and EMF objects: SDO objects are simply EMF objects that also implement the SDO DataObject interface. In fact, the implementation of these additional methods are implemented by delegation to the core EMF ones.

### Creating SDO instances

Now that we have a model of our data objects, we can build instances of Employee and set various properties on them. As stated previously, we will use the EMF API to maximize performance.
EObject eObject = EcoreUtil.create(employeeClass);

// Note: we could cast the object to DataObject,
// but chose to use EObject APIs instead.
eObject.eSet(employeeNameFeature, name);
eObject.eSet(employeeNumberFeature, new Integer(number));

We can then "link" employees together using the "employees" reference, for example:

((List)bigBoss.eGet(employeeEmployeesFeature)).add(terence);

Once we've created the data objects, we need to attach them to the data graph. We do this by calling the data graph's setRootObject() method, passing the data object we want to be at the root, which in this case is Employee The Boss.

EDataGraph employeeGraph = SDOFactory.eINSTANCE.createEDataGraph();

employeeGraph.setERootObject(rootObject);

One last thing to do before returning the data graph is to start logging changes. Before changes are going to be made to a data graph, beginLogging() should be called on its change summary if you want to use SDO's capabilities. What this does is basically starts listening for changes after clearing all previous changes.

// Call beginLogging() so that the Change Summary is
// populated when changes are applied to the Data Graph.
// The DMS should call beginLogging() and endLogging(),
// not the client.
employeeGraph.getChangeSummary().beginLogging();

Another task of the DMS (as defined in the EmployeeDataMediator interface) is to update backend data sources based on a data graph provided by the SDO client.

The DMS: Updating source
To update backend data sources, DMSs should use the powerful features of SDO, more specifically its change summary. There are various ways to use the a data graph's change summary. In this example, we look at all the data objects referenced from the change summary tree and get the new data objects from there.

Listing 4. DMS updating backend according to a data graph

```java
/**
 * Update the DMS's backend data to reflect changes
 * in the data graph.
 * Since this DMS has no actual backend data and therefore
 * has nothing to update, we will just navigate
 * the change summary and report (print) what's changed.
 */
public void update(DataGraph dataGraph)
{
    ChangeSummary changeSummary = dataGraph.getChangeSummary();

    // Call endLogging to summarize changes.
    // The DMS should call beginLogging() and endLogging(),
```
// not the client.
changeSummary.endLogging();

// Use SDO ChangeSummary's getChangedDataObjects() method.
List changes = changeSummary.getChangedDataObjects();
for (Iterator iter = changes.iterator(); iter.hasNext();)
{
    DataObject changedObject = (DataObject)iter.next();
    System.out.print("DMS updating "+
    changedObject.getString("name");

    for (Iterator settingIter = changeSummary.getOldValues(
        changedObject).iterator(); settingIter.hasNext();)
    {
        ChangeSummary.Setting changeSetting =
            (ChangeSummary.Setting)settingIter.next();
        Property changedProperty = changeSetting.getProperty();
        Object oldValue = changeSetting.getValue();
        Object newValue = changedObject.get(changedProperty);
        System.out.print(" (changed: "+
            changedProperty.getName() +
            " from " + oldValue + " to " + newValue + ")");
        // If not a simple example, we could update the backend here.
    }
    System.out.println();
}

In this example, no backend update takes place. In reality, backend updates would take place in this method.

The first thing the DMS does when it gets back a data graph from the client for backend update is call endLogging() on the data graph's change summary. Doing this turns off change recording, thereby providing a summary of the modifications that were made to a data graph since beginLogging() was called (typically since its creation). It is in a format that allows the DMS to update the backend data sources efficiently and incrementally. Modifications in the change summary are of three types:

- **Object changes** contain references to data objects in the data graph whose properties have been modified, along with the property that was changed and the old value for that property. The old value can be used by the DMS to make sure backend data hasn't been modified by someone else in the meantime.
- **Object creations** contain the data objects that were added to the data graph. These objects represent new data that needs to be added to the backend data structure.
- **Object deletions** contain the data objects that were deleted from the data graph. These objects represent data that needs to be removed from backend data structure.

Notice that we used the standard SDO API to inspect the data graph's changes. We could, however, have used the EMF ChangeDescription API (instead of SDO's ChangeSummary). In this example, updating the value of simple attributes, the performance impact would not be significant. For other cases, such as when changing multiplicity-many properties, using the EMF API could improve performance drastically. For example, say we remove one employee from a list of a few hundred employees. In this case, the ChangeSummary only provides access to the old value, that is the old list of a few hundred employees. EMF's ChangeDescription interface, on the other hand,
also provides more precise information, such as "remove employee at some index," which would be much more useful.

Note also that in this example, there are only object changes in the change summary, no removals or additions. If you play with the SDO implementation and remove objects from the data graph, you'll notice elements of type objectsToAttach. This is actually the EMF ChangeDescription's name for object deletions. They are the data objects that were deleted and need to be attached back to the graph in case of rollback, which is EMF's view of the change. So to summarize, objectsToAttach == deleted objects.

**Debugging the application**

If you set the debug variable to true in the sample application, it enables calls like the one below, which lets you see the serialized version of the data graph.

```java
((EDataGraph) dataGraph).getDataGraphResource().save(System.out, null);
```

You can also use the Eclipse debug environment. For example, we suggest you set a breakpoint in SDOClient.java, line 110, and debug SDOClient (as a Java application). Then, in the debug perspective, you can see the data graph in memory (under Variables), with its data objects (The Boss, Terrence Shorter, etc.), as shown in Figure 4.
Figure 4. Looking at the data objects in debug mode

This way also lets you see the change summary, as shown in Figure 5.
Figure 5. Looking at the change summary in debug mode

The screen captures above look complex and you may not find them useful now but you might want to come back to them when you are debugging your SDO application and looking for the contents of your data objects and change summaries.

Conclusion

In this article, we've provided an overview of SDO and its capabilities. We've showed a sample application that uses some of SDO's capabilities. See the SDO API documentation under the Eclipse help system for further reference. The specification is still evolving and being enhanced. For example, SDO 1.0 focused on the SDO client's perspective and didn't specify a DMS API. SDO is currently being standardized through the JCP, so watch out for announcements. Because SDO is so flexible, there will be lots of decisions you will have to make when you design your SDO application. These decisions will impact reusability and performance. So you should really think of the usage patterns and characteristics of your application data before you code.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j-sdoSample.zip</td>
<td>13KB</td>
</tr>
</tbody>
</table>
Related topics

• You can download the source code used in this article by clicking on the Code icon at the top or bottom of this article (or see the Download section).
• The IBM Development Package for Eclipse bundles Eclipse with the latest Java runtimes (Linux and Windows) from IBM.
• IBM's reference implementation of SDO 1.0 is packaged with the Eclipse Modeling Framework (EMF). You'll find articles, FAQs, and a newsgroup on the EMF home page.
• Follow the standardization of SDO with JSR-235 on the JCP Web site.
• "Using Service Data Objects with Enterprise Information Integration technology" (developerWorks, July 2004) shows an example of using SDO.
• "Creating JSF applications that access data using Web Data Objects" (developerWorks, March 2004) shows another example of using SDO.
• "Get started with XPath" (developerWorks, May 2004) is for you if you want to learn XPath.
• Eclipse Modeling Framework (Addison-Wesley; 2003), by F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T.J.Grose, is an excellent reference on all aspects of EMF.

© Copyright IBM Corporation 2004
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)