The Tuscany SDO Java Project

March, 2006

This document provides a high-level overview of the Java SDO (Service Data Objects) subproject of the Tuscany incubator project at Apache.org.

Table of Contents

1. Overview

The SDO Java project is a subproject of Tuscany (http://incubator.apache.org/tuscany/) intended to provide a Java implementation of the SDO 2 specification (http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-sdo/SDO_Specification_Java_V2.01.pdf).

The project's code base includes the following:

- Dynamic data object support
- Basic static code generation (generator patterns still subject to change)
- Most Helper classes either partially or fully implemented (XMLHelper, XSDHelper, DataFactory, CopyHelper, EqualityHelper)
- Minimal ChangeSummary support (only in DataGraph context - ChangeSummary attributes TBD)
- Very limited example programs

TBD…

2. Build Environment Setup

2.1 Tuscany Build Environment Setup

SDO 2 is a subproject of the Tuscany project. If you check out and build the whole Tuscany Java project, you will have also built the SDO 2 subproject. If you want to work with the SDO 2 project, without the rest of Tuscany, proceed to section 2.2.

To checkout and build Tuscany, proceed with the following steps:

1. Download and install maven 2 (build tool) from http://maven.apache.org/
2. Download and install subversion (version control system) from http://subversion.tigris.org/
3. Make sure 'mvn' and 'svn' commands are in your PATH environment variable.
4. Check out Tuscany open source project from Apache
    Commands:
        md <local tuscany dir>
        cd <local tuscany dir>
        svn co https://svn.apache.org/repos/asf/incubator/tuscany/java

5. Run "mvn" under <local tuscany dir>/java directory to install POM files from the root project to the local repository and to build the entire project
    Commands:
        cd <local tuscany dir>/java
        mvn
6. If you are using Eclipse for development, proceed to step 6 in section 2.2 to run "mvn eclipse:eclipse".

Note: If the mvn command completed successfully, you will see BUILD SUCCESSFUL in the output.
External resources are at times unavailable. It may be necessary to run "mvn" again at a later time.
If you are taking time to reply to firewall prompts, this can cause some requests to time out. Set up the firewall to permit the action without prompting.

2.2 SDO Build Environment Setup

If you want to work with the SDO 2 project alone, without the rest of Tuscany, proceed with the following steps.

1. Download and install maven 2 (build tool) from http://maven.apache.org/
2. Download and install subversion (version control system) from http://subversion.tigris.org/
3. Make sure 'mvn' and 'svn' commands are in your PATH environment variable.
4. Check out the SDO open source projects from Apache.
    Commands:
        md <local tuscany dir>
        cd <local tuscany dir>
        svn co –N https://svn.apache.org/repos/asf/incubator/tuscany/java
        cd java
        svn up sdo
        svn up –N spec
        cd spec
        svn up sdo

5. Run "mvn" under <local tuscany dir>/java directory to install POM files from the root project to the local repository
    Commands:

cd <local tuscany dir>/java
mvn –N
cd spec
mvn –N
cd ../sdo
mvn –N (alternatively, run without the –N option – see Note below)

6. Build, or rebuild, the individual SDO projects

sdo.spec project

Commands:

cd <local tuscany dir>/java/spec/sdo
mvn
mvn eclipse:eclipse (optional: Run this command if you are using Eclipse for development.)

sdo.impl project

Commands:

cd <local tuscany dir>/java/sdo/impl
mvn
mvn eclipse:eclipse (optional: Run this command if you are using Eclipse for development.)

sdo.tools project

Commands:

cd <local tuscany dir>/java/sdo/tools
mvn
mvn eclipse:eclipse (optional: Run this command if you are using Eclipse for development.)

Note: You can build both sdo.impl and sdo.tools in one step by running mvn in <local tuscany dir>/java/sdo.

Note: If the mvn command completed successfully, you will see BUILD SUCCESSFUL in the output and sdo-api-SNAPSHOP.jar is created under <local tuscany dir>/java/spec/sdo/target directory.
External resources are at times unavailable. It may be necessary to run "mvn" again at a later time.
If you are taking time to reply to firewall prompts, this can cause some requests to time out. Set up the firewall to permit the action without prompting.

3. SDO Project Structure

The SDO project is divided into three parts:

1. **sdo.spec** contains the SDO (commonj) interfaces defined and provided by the SDO 2 specification.
2. **sdo.impl** provides the runtime implementation of the SDO interfaces.
3. **sdo.tools** contains import and generator tools.

The main source code in each of these subprojects is located in the directory src/main/java, and if applicable, test (example) classes are located in src/test/java. The directory src/test/resources contains any data files needed by the test programs.

3.1 sdo.spec (https://svn.apache.org/repos/asf/incubator/tuscany/java/spec/sdo)

This project contains the interfaces provided with the SDO 2 specification. It is essentially an unzipped copy of the SDO Java API sources zip file available at http://ftpna2.bea.com/pub/downloads/SDO_20_Source.zip, but with some errata corrections and a Tuscany-specific implementation of class HelperProvider.

The abstract class, HelperProvider, is used to obtain specific default helpers and other implementation-specific objects used by the Java implementation of SDO. In the Tuscany implementation of this class, there are two ways to specify the implementation of the HelperProvider class.

1. Set a System Property named "commonj.sdo.impl.HelperProvider" equal to the fully qualified class name of the implementation class (e.g. "commonj.sdo.impl.HelperProvider=org.apache.tuscany.sdo.help.HelperProviderImpl").

2. In your own jar file, create a text file called "META-INF/services/commonj.sdo.impl.HelperProvider". In this text file, specify the fully qualified custom HelperProvider implementation class (e.g. org.apache.tuscany.sdo.help.HelperProviderImpl).

In the event that both 1 and 2 are specified, the System Property will take precedence over the text file.

The Tuscany default helper provider implementation class is org.apache.tuscany.sdo.helper.HelperProviderImpl (in the sdo.impl project) and is registered using the second technique (services file), as described in the following section.

3.2 sdo.impl (https://svn.apache.org/repos/asf/incubator/tuscany/java/sdo/impl)

The sdo.impl subproject contains a test package under src/test/java (see section 5 for details) and the following implementation packages under src/main/java:

package org.apache.tuscany.sdo

Contains a few interfaces used by some of the implementation classes in org.apache.tuscany.sdo.impl. (Note: this package is subject to further cleanup.)

package org.apache.tuscany.sdo.helper

This package contains implementations of the "helper" interfaces defined in the commonj.sdo.helper package (in the sdo.spec project). Each helper interface in commonj.sdo.helper has a corresponding implementation class in this package. The name of each helper class is the same as the corresponding interface, only with the suffix "Impl"

appended. For example class org.apache.tuscany.sdo.helper.TypeHelperImpl implements the interface commonj.sdo.TypeHelper.

The implementation class org.apache.tuscany.sdo.helper.HelperProviderImpl is used to bootstrap an implementation of the default INSTANCEs of the SDO helper (see class commonj.sdo.impl.HelperProvider in sdo.spec). This implementation creates instances of the other helper implementation classes in this package and is registered using the services file src/main/resources/META-INF/services/commonj.sdo.impl.HelperProvider.

package org.apache.tuscany.sdo.impl

This package contains the majority of the SDO runtime implementation code. This includes implementations of all of the commonj.sdo interfaces (see sdo.spec), including several implementations of the DataObject interface. The design and implementation of the most important classes in this package are described in section 3 (below).

package org.apache.tuscany.sdo.util

Contains some utility classes used by the implementation. One class, SDOUtil, is particularly important. It provides some useful static utility functions which are not included in the SDO specification itself. Although these are not "standard" APIs, use of them is recommended, as opposed to resorting to low-level implementation-specific APIs. The intent of this class is to encapsulate, in a relatively clean way, common functions that are needed, and can potentially be proposed for addition to the specification in a future version of SDO.

3.3 sdo.tools (https://svn.apache.org/repos/asf/incubator/tuscany/java/sdo/tools)

This project will contain (command line) tools, such as SDO model importers and generators (Java code, XML schema, etc.). Currently however, there is only a single tool, a Java code generator implemented in class org.apache.tuscany.sdo.generate.JavaGenerator. This generator can be used to generate static SDO data objects and is described in more detail in section 4.

The sdo.tools project also contains a test program and sample generated model located in src/test/java and src/test/resources respectively (see section 5 for more details).

3.4 Dependency Jars

The sdo.impl project requires the following EMF (Eclipse Modeling Framework - www.eclipse.org/emf) runtime jars to build:

- **emf-common-2.2.0-SNAPSHOT.jar** – some common framework utility and base classes
- **emf-ecore-2.2.0-SNAPSHOT.jar** – the EMF core runtime implementation classes (the Ecore metamodel)
- **emf-ecore-change-2.2.0-SNAPSHOT.jar** – the EMF change recorder and framework
- **emf-ecore-xmi-2.2.0-SNAPSHOT.jar** – EMF's default XML (and XMI) serializer and loader
- **xsd-2.2.0-SNAPSHOT.jar** – the XML Schema model

The sdo.tools project also requires the EMF code generator framework jars:

- **emf-codegen-2.2.0-SNAPSHOT.jar** – template-based codegen framework (JET – Java Emitter Templates)
- **emf-codegen-ecore-2.2.0-SNAPSHOT.jar** – the EMF code generator
- **emf-common-2.2.0-SNAPSHOT.jar** – some common framework utility and base classes
- **emf-ecore-2.2.0-SNAPSHOT.jar** – the EMF core runtime implementation classes (the Ecore metamodel)
- **emf-ecore-change-2.2.0-SNAPSHOT.jar** – the EMF change recorder and framework
- **emf-ecore-xmi-2.2.0-SNAPSHOT.jar** – EMF's default XML (and XMI) serializer and loader
- **xsd-2.2.0-SNAPSHOT.jar** – the XML Schema model

These are simply Maven-friendly versions of corresponding jar files/plugins obtained from Eclipse. SNAPSHOT maps to an EMF weekly integration build (for example, I200602160000).

4. Runtime Implementation

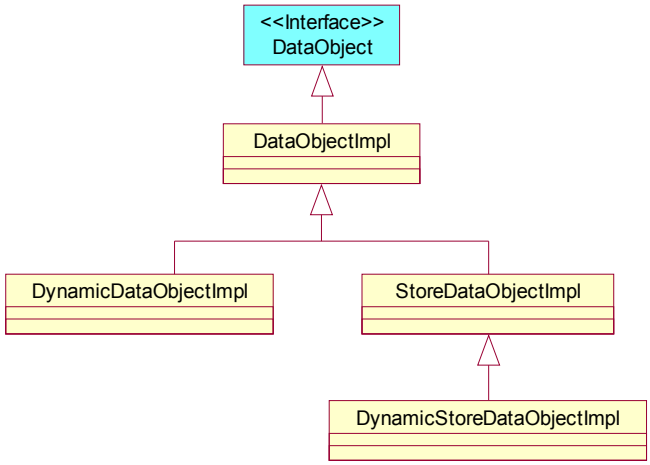
The primary SDO runtime implementation classes are located in the package org.apache.tuscany.sdo.impl and consist of the following:

1. DataObject implementation classes
2. Implementation of the SDO metamodel interfaces: Type and Property
3. ChangeSummary and DataGraph implementations

The implementation of the SDO runtime is based on and leverages the EMF runtime model (i.e., EObject and the Ecore metamodel – refer to documentation at www.eclipse.org/emf). It subclasses and specializes the Ecore metamodel, and provides its own DataObject-tuned implementation(s) of the EObject interface. The design is described in more detail in the following sections.

4.1 DataObject implementation classes

SDO provides several DataObject implementation classes as shown in the following diagram:



Class DataObjectImpl is the most important. It provides a complete base implementation of the SDO DataObject interface. It extends from the EMF base class BasicEObjectImpl, which provides the "scaffolding" needed to easily implement an EObject, but without allocating any storage itself.

DataObjectImpl provides the DataObject implementation while allocating only the minimum storage overhead needed to be a data object (e.g., container pointer and feature, change recorder). It does not, however, allocate any storage for the actual properties of the data object. It instead requires subclasses for this purpose. For example, statically generated SDOs (see section 4) directly or indirectly extend from this class, providing their own storage in generated instance variables.

The subclass, DynamicDataObjectImpl serves as a concrete implementation class for dynamic data objects. It is the default implementation class used when creating dynamic data objects using the DataFactory.create() method, for example. DynamicDataObjectImpl provides efficient data storage using a dynamically allocated settings array.

StoreDataObjectImpl and DynamicStoreDataObjectImpl provide a delegating implementations for DataObjects that implement their own storage management using a store (see EMF's EStore interface) implementation class. StoreDataObjectImpl is used in conjuction with the "-storePattern" generator option (see section 4), while DynamicStoreDataObjectImpl, as its name implies, is used for dynamic store-based instances.

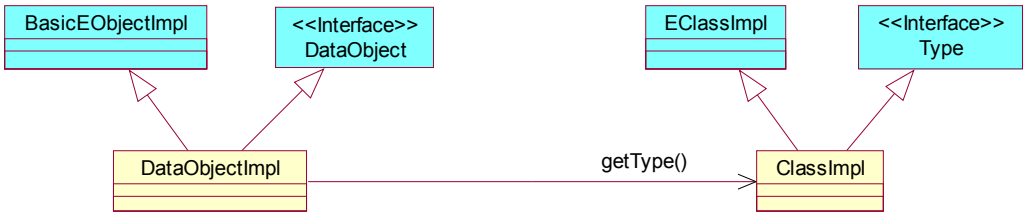
4.2 Type and Property implementation classes

The SDO implementation provides three implementations of the interface Type, one for each of the following three kinds of types: classes, simple data types, and enumerations.

1. class ClassImpl extends EClassImpl implements Type
2. class DataTypeImpl extends EDataTypeImpl implements Type
3. class EnumImpl extends EEnumImpl implements Type

For example, class org.apache.tuscany.sdo.impl.ClassImpl extends form the corresponding Ecore class, EClassImpl, and mixes in the SDO interface commonj.sdo.Type. All the Type methods are implemented by calls to super.

With this approach, a data object's Type, returned from DataObjectImpl.getType(), and its EClass, returned by DataObjectImpl.eClass(), are the same underlying meta object. This allows the SDO implementation to leverage any appropriate base functionality without any performance overhead. The arrangement is shown in the following diagram:

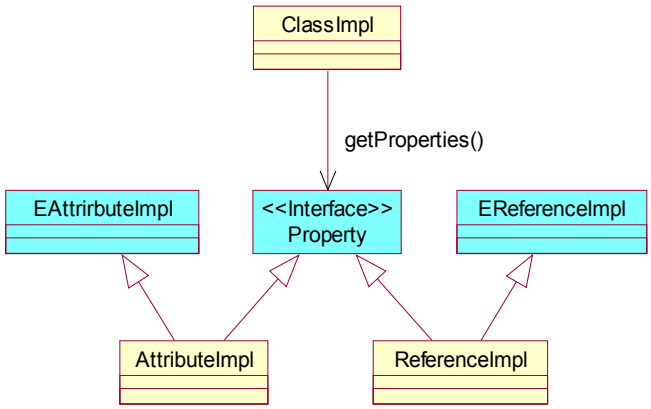


The implementation of the SDO Property interface follows a similar pattern. Two implementation classes, subclasses of corresponding Ecore classes, mix in the Property interface:

1. class AttributeImpl extends EAttributeImpl implements Property
2. class ReferenceImpl extends EReferenceImpl implements Property

As with the Type implementation classes, these classes call methods on super to implement the mixed-in Property methods.

The following diagram illustrates the design:



As shown, the getProperties() method in ClassImpl (i.e., of the SDO Type interface) returns a set properties whose implementation classes are also EAttributes or EReferences, and since ClassImpl, extends EClassImpl (as shown in the previous diagram), these are in fact the same objects as those returned by the EClass.getEAllStructuralFeatures() method. The two metamodels are one and the same, making the implementation of many of the SDO APIs trivial calls to the base class.

4.3 ChangeSummary and DataGraph implementation classes

TBD…

5. Static Code Generator

The SDO static code generator is a command line tool for generating Java source code (static SDOs) for DataObjects defined in an XML Schema. It is implemented by the class org.apache.tuscany.sdo.generate.JavaGenerator in the sdo.tools project. The generator is used as follows:

Usage arguments:

[-targetDirectory <target-root-directory>]
[-javaPackage <base-package-name>]
[-prefix <prefix-string>]
[-sparsePattern | -storePattern]
[-noInterfaces] [-noContainment] [-noNotification] [-arrayAccessors] [-noUnsettable]
<xsd-file> | <wsdl-file>

For example:

java JavaGenerator somedir/somefile.xsd

Options:

-**targetDirectory** Generates the Java source code in the specified directory. By default, the code is generated in the same directory as the input xsd or wsdl file.

-**javaPackage** Overrides the Java package for the generated classes. If not specified, a default package or one specified with an sdoJava:package annotation on the <schema> element in the xsd file (see SDO specification for details) is used for the java package.

-**prefix** Specifies the prefix string to use for naming the generated factory. For example "-prefix Foo" will result in a factory interface with the name "FooFactory".

-**sparsePattern** For SDO metamodels that have classes with many properties of which only a few are typically set at runtime, this option can be used to produce a space-optimized implementation (at the expense of speed).

-**storePattern** This option can be used to generate static classes that work with a Store-based DataObject implementation. It changes the generator pattern to generate accessors which delegate to the reflective methods (as opposed to the other way around) and changes the DataObject base class to org.apache.tuscany.sdo.impl.StoreDataObjectImpl. Note that this option generates classes that require a Store implementation to be provided before they can be run.

-**noInterfaces** By default, each DataObject generates both a Java interface and a corresponding implementation class. If an SDO metamodel does not use multiple inheritance (which is always the case for XML Schema derived models), then this option can be used to eliminate the interface and to generate only an implementation class.

-**noNotification** This option eliminates all change notification overhead in the generated classes. Changes to DataObjects generated using this option cannot be recorded, and consequently the classes cannot be used with an SDO ChangeSummary or DataGraph.

-**noContainment** Turns off container management for containment properties. DataObject.getContainer() will always return null for data objects generated with this option, even if a containment reference is set. Setting a containment reference will also not automatically remove the target object from its previous

container, if it had one, so it will need to be explicitly removed by the client. Use of this option is only recommended for scenarios where this kind of container movement/management is not necessary.

-**arrayAccessors** Generates Java array getters/setters for multiplicity-many properties. With this option, the set of "standard" JavaBean array accessor methods (e.g., Foo[] getFoo(), Foo getFoo(int), int getFooLength(), setFoo(Foo[]), and void setFoo(int, Foo)) are generated. The normal List-returning accessor is renamed with the suffix "List" (e.g., List getFooList()). The array returned by the generated method is not a copy, but instead a pointer to the underlying storage array, so directly modifying it can have undesirable consequences and should be avoided.

-**noUnsettable** By default, some XML constructs result in SDO property implementations that maintain additional state information to record when the property has been set to the "default value", as opposed to being truly unset (see DataObject.isSet() and DataObject.unset()). The SDO specification allows an implementation to choose to provide this behavior or not. With this option, all generated properties will not record their unset state. The generated isSet() methods simply returns whether the current value is equal to the property's "default value".

5.1 Generator Patterns

The DataObject interface generation pattern is as described in the SDO specification (see *Java Interface Specification* section). The SDO specification does not define a factory pattern for efficient construction of static SDOs, which is however provided by the Tuscany implementation. The generated SDO Factory interface conforms to the following pattern:

```
public interface <prefix>Factory {
  <Type1> create<Type1>();
  <Type2> create<Type2>();
  ...
  <prefix>Factory INSTANCE = <default_factory_impl>;
}
```

A generated factory corresponds to an SDO Type namespace uri (see commonj.sdo.Type.getURI) with one create() method for each SDO Type in the namespace. The <prefix> of the factory name is derived from the uri. An instance of the factory is available using the INSTANCE field in the interface.

Using the static factory, a DataObject might be created as follows:

```
Quote aQuote = StockFactory.INSTANCE.createQuote();
... // do something with aQuote
```

The generated implementation of each create() method simply constructs an instance of the corresponding type like this:

```
  public Quote createQuote() {
    QuoteImpl quote = new QuoteImpl();
    return quote;
  }
```

In addition to these generated type-specific create<Type>() methods, the generated factory implementation class also includes a generated reflective create() method that, given an SDO Type, efficiently dispatches to the correct type-specific create() method. The reflective create() method is called by the implementation of the SDO commonj.sdo.helper.DataFactory interface.

6. Test/Example Programs

The SDO project does not include any proper sample programs at this time (any volunteers?) but it does include a number of JUnit test cases, some of which serves as good examples of how to use SDO APIs to perform various tasks.

The following tests are particularly good SDO examples included in the sdo.impl project:

- **SimpleDynamicTestCase** – This program uses the SDO XSDHelper.define() method to register a simple XML Schema based model (simple.xsd). It then instantiates a DataObject instance (type Quote), initializes several of its properties, and then serializes the instance to an XML stream.

- **MixedTypeTestCase** – This program shows how to uses the Sequence API to create an XML instance which mixes arbitrary text within the DataObject's XML structure. It uses the XML schema complexType (MixedQuote) in mixed.xsd to define the SDO model.

- **OpenTypeTestCase** – Uses an XML Schema complexType with a wildcard (xsd:any) to illustrate how to work with and manipulate an SDO open type. The type OpenQuote in open.xsd is used for this example.

- **SimpleCopyTestCase** – Uses the SDO CopyHelper to create shallow and deep copies of the simple Quote model from SimpleDyanmicTest.

- **SimpleEqualityTestCase** – Uses the SDO EqualityHelper to perform deep and shallow equality checks.

- **ChangeSummaryTestCase** – Creates a data graph with an instance of type Quote (in simple.xsd) as the root object. It then turns on change logging and makes a number of changes to the graph. Finally, it turns off change logging and serializes the data graph.

- **XSDHelperTestCase** – This program shows how the XSDHelper.define() method can be called multiple times with the same schema. The second (and subsequent) call simply returns the previously defined list of types.

The following is in the sdo.tools project:

- **SimpleStaticTestCase** – This test performs the same function as SimpleDynamicTest, above, only using a generated version of the simple.xsd model. The generated model has been pre-generated (with default options) in the directory src/test/resources, but it can be regenerated using the JavaGenerator, possibly with different generator options (e.g., -noInterfaces or –sparsePattern), if desired.