Service Data Objects in SOA Projects: Practical Experience

Andrei Shakirin
Talend GmbH, Application Integration Division
Bonn, Germany

Keywords: Service Data Objects, Data Graph Synchronization, Distributed Applications, JPA, EclipseLink

Introduction

One of the common challenges of stateless service oriented design is merging of complex data structures between client and service and keeping the data graph in consistent state. A possible solution of this problem is using Service Data Objects (SDO) for communication between clients and service.
The presentation summarizes experience and lessons learned regarding using of Service Data Objects (SDO) in concrete SOA projects. The first part of presentation will introduce Service Data Objects and explain basic principles of SDO approach. It contains definition and purposes of SDO; represents Data Object, Data Graph and Data Access Service abstractions; makes a short overview of existing SDO implementations.

Second part of presentation provides typical SDO scenarios and use cases. Either application needs to track changes in the complex data structure or remote client makes a fine grained changes in interconnected data graph and will merge data on the service side for further processing and persistence – in all such cases architect can think about SDO approach as a possible solution. Live demo based on open source SDO implementation will demonstrate SDO approach in the practice.

The third part of presentation will show using of SDO in one concrete project developed last year for Greenpeace UK. The project architecture represents compound, business and basic services layers and shows the SDO role in communication between consumers and services. As far as project was developed in close collaboration with EclipseLink team, some aspects of integration the SDO EclipseLink implementation with ESB platforms are also covered in this part.
Interesting part of this project was integration between SDO and JPA to persist complex data graph into the database. Presentation shows two possible SDO persistency solutions and explains the actual project choice.
The main idea of this presentation is to show the practical using of SDO approach in service oriented projects and to discuss one more interesting design pattern

SDO and other Data Application Frameworks

What is the added value of SDO in comparison with other data application frameworks? I can highlight three main things:

1. Unifying data programming across different data formats and data source types (RDBs, EJBs, Web Services, Messaging Providers, JCA, POJOs)
2. Incorporating a number of application patterns and best practices (change history, disconnected programming model, compositor)

3. Enabling application, tools and frameworks to more easily query, view, update, bind, transfer data and keep it consistent

In order to show SDO position, following table compares it with other data application frameworks:

SDO and JAXB	Similar in sense of model Java-XML binding, but XML is not only one data source for SDO. SDO provides uniform access for various types of data. There is a bridge between SDO and JAXB models.
SDO and JPA	Similar in sense of simplifying java data programming and abstract from specific technology. Both frameworks provide a context and manage attached data objects. JPA is concentrated on data persistency in RDBs, SDO is more general and represents data that can flow between any application tier.
SDO and EMF	Very similar in purpose, SDO can be considered as a thin layer over EMF. IBM reference SDO implementation is EMF based.
SDO and SCA	Complementary specs. SCA defines assembly of service components into business solution. SDO concentrates on the data model representation. Can be used together and separated.

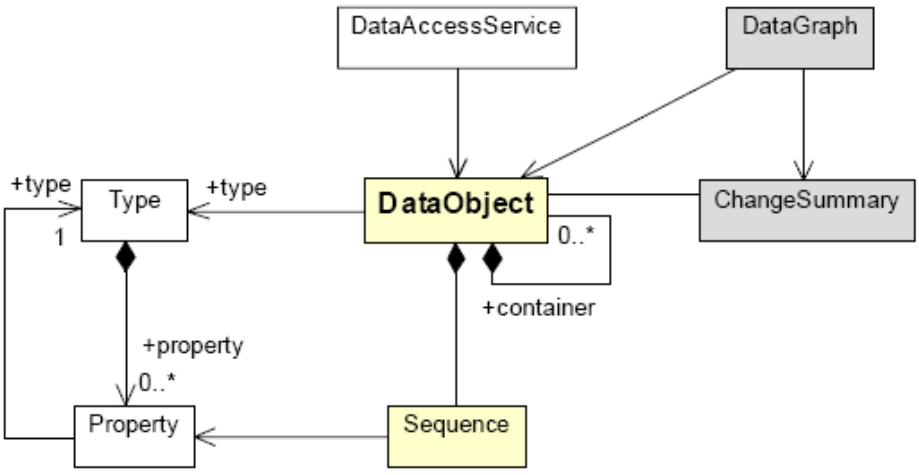
Table. 1: Comparison of SDO and other Data Application Frameworks

SDO Definition

SDO can be explained using three basic definitions:

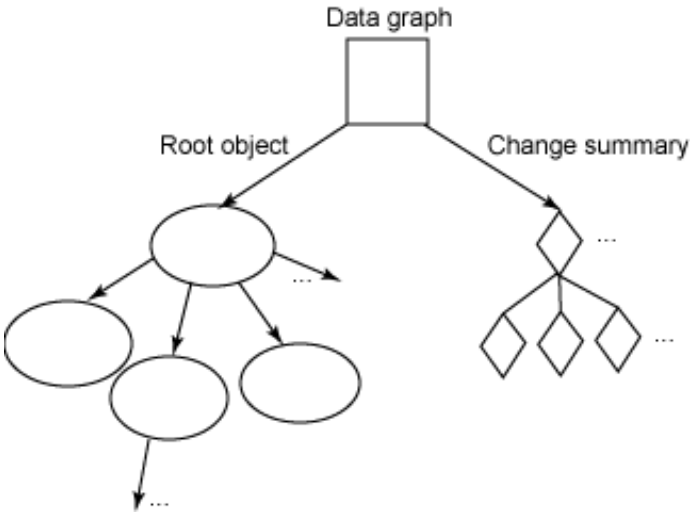
- DataObject
- DataGraph
- Data Access Service

DataObject represents a business data. It holds a set of named properties, each of which contains either a simple data-type value or a reference to another Data Object (simple, containment, read-only, multi-valued, open). The Data Object API provides a dynamic and static data API for manipulating these properties.



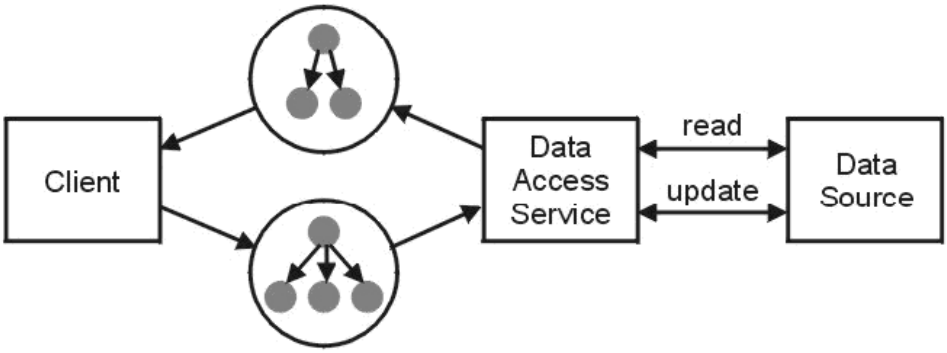
Illustartion. 1: DataObject

DataGraph is envelope for Data Objects, and it is the normal unit of transport between components. DataGraph is a closed tree (cannot have a circles, exactly one parent). Data graphs can track changes made to the Data Objects.



Illustartion. 2: DataGraph

Data Access Service populates data graphs from data sources and commits changes to data graphs back. Uses disconnected data architecture.



Illustartion. 3: Data Access Service

Additional aspects of SDO specification are:

- Dynamic and Static APIs
- Change Summary (including XML representation)
- Validation, Contrains (via metadata and extensions) and Relationship Integrity: cardinality, closure, ownership semantic and inverses
- Helpers API
- Sequences, open properties, containment and non-containment references

Today the following SDO implementations are available:

- EclipseLink
- Apache Tuscany (integrated with Apache CXF)
- SCA and SDO for PHP (PECL extension)
- Rogue Wave Software (Hydro SDO)
- Xcalia
- IBM WebSphere (Ecore based)

Problem Definition

Service Oriented applications are normally stateless. It makes them more scalable and easy in design and implementation. However the statelessness has some serious challenges:

1. Heavily interconnected objects modified on the client side become hard to merge on the server side
2. It is not always possible to merge data graph automatically and even consistently
3. Because the transport of whole graph is too expensive or not desired, service facade is extended with additional methods to manage each particular part of graph. The server likes to know in advance which subgraph should be proceed (causes „God facades“: getCustomerWithAddress() / setCustomerWithAddress() which are redundant and very difficult to maintain).

SDO provides a solution for these problems.

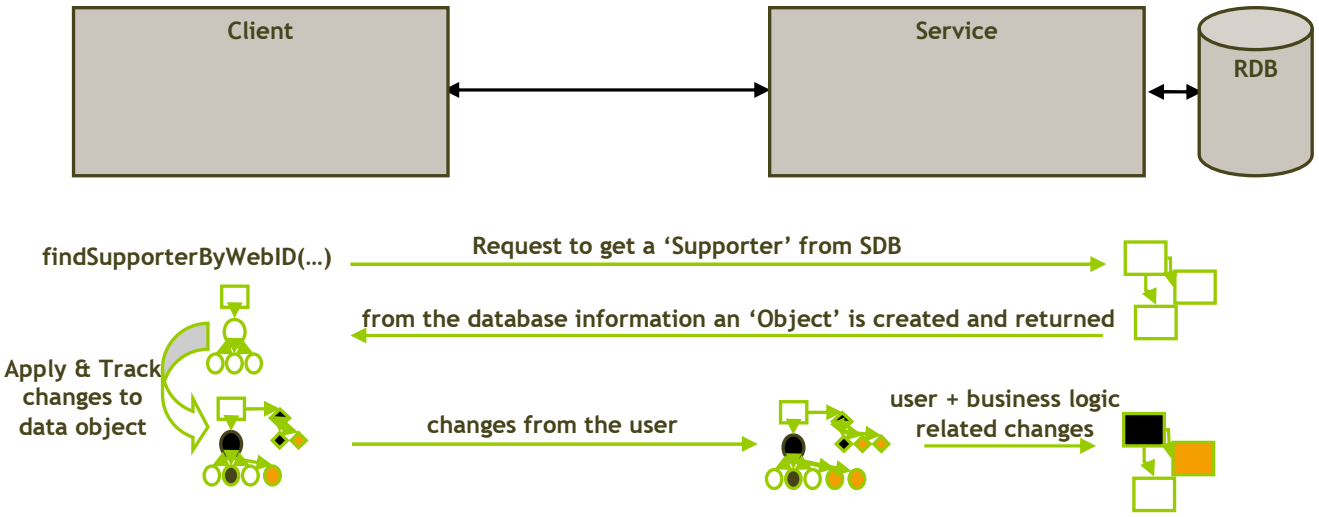
Solution Using SDO

How SDO can help to solve formulated problem? SDO provides very useful, rich and standard mechanism to track, transport and merge the changes in Data Graph.
Typical SDO solution looks like the following:

1. Client requests a data graph from the service.
2. Service creates SDO model on the base of some metadata (it can be XMLSchema, DB schema, UML) and populates it with data from the data source (relation database, JMS queue, EJB).
3. Service serializes the graph or part of the graph using SDO helper and sends it to the client in standardized format.
4. Client deserializes the data graph and processes it: reads, searches for objects, makes updates, deletes and inserts the new objects. Change summary tracks all client changes.
5. Client serializes modified data graph inclusive change summary and sends it back to server.
6. Service deserializes the data graph inclusive change summary and has all information about the changes. It is trivial now to merge data on the server side consistently, validate and update data into backend storage.

As far as SDO specification is language neutral, server and client can even be implemented using different programming languages and technologies.

The following figure illustrates SDO solution for data graph merging in context of concrete project:



Illustartion. 4: Using SDO to merge Data Graph between client and service

SDO in Java

How SDO looks like in Java? Here is the listing of a small sample that defines the metamodel based on XMLSchema, loads data graph as XML file and serializes it:

```
// 1. Load metamodel
XSDHelper.INSTANCE.define(HelloWorldSDO.class.getResourceAsStream("/customer.xsd"), null);

// 2. Load data graph
XMLDocument load = XMLHelper.INSTANCE.load(HelloWorldSDO.class.getResourceAsStream("/customer-
data.xml"));

// 3. Get the root object
CustomerType rootObject = (CustomerType)load.getRootObject();

// 4. Print the root object property
System.out.println("FirstName: " + rootObject.getPersonalInfo().getFirstName());

// 5. Print the data graph
System.out.println(XMLHelper.INSTANCE.save((DataObject)rootObject, null, "Customer"));
```

This code produces the following output:

```
FirstName: Jane
<?xml version="1.0" encoding="UTF-8"?>
<Customer xmlns:ns1="http://www.example.org/customer-example"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="ns1:customer-type">
   <personal-info>
      <first-name>Jane</first-name>
      <last-name>Doe</last-name>
      <ns1:gender>F</ns1:gender>
   </personal-info>
   <ns1:contact-info>
      <billing-address>
         <street>123 Any Street</street>
         <street>Suite 1</street>
         <city>Ottawa</city>
         <state>ON</state>
         <zip-code>A1B 2C3</zip-code>
      </billing-address>
      <shipping-address>
         <street>456 Another Street</street>
         <city>Raleigh</city>
```

```
         <state>NC</state>
         <zip-code>12345</zip-code>
      </shipping-address>
      <ns1:phone-number number-type="work">(613) 555-1111</ns1:phone-number>
      <ns1:phone-number number-type="cell">(613) 555-2222</ns1:phone-number>
   </ns1:contact-info>
   <changeSummary logging="false" xmlns:sdo="commonj.sdo"/>
</Customer>
```

The second sample additionally activates change tracking and makes some changes in data graph:

```
// 1. Load metamodel
XSDHelper.INSTANCE.define(HelloWorldSDO.class.getResourceAsStream("/customer.xsd"), null);

// 2. Load data graph
XMLDocument load = XMLHelper.INSTANCE.load(HelloWorldSDO.class.getResourceAsStream("/customer-
data.xml"));

// 3. Get the root object
CustomerType rootObject = (CustomerType)load.getRootObject();

// 4. Print the root object property
System.out.println("FirstName: " + rootObject.getPersonalInfo().getFirstName());

// 5. Activate change tracking and made some changes
rootObject.getChangeSummary().beginLogging();
rootObject.getContactInfo().getBillingAddress().setCity("updatedCity");
rootObject.getContactInfo().getPhoneNumber().remove(0);

PhoneNumber number = (PhoneNumber)DataFactory.INSTANCE.create(PhoneNumber.class);
number.setValue("12345");
rootObject.getContactInfo().getPhoneNumber().add(number);
rootObject.getChangeSummary().endLogging();

// 5. Print the data graph
System.out.println(XMLHelper.INSTANCE.save((DataObject)rootObject, null, "Customer"));
```

Now the output is:

```
FirstName: Jane
<?xml version="1.0" encoding="UTF-8"?>
<Customer xmlns:ns1="http://www.example.org/customer-example"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="ns1:customer-type">
   <personal-info>
      <first-name>Jane</first-name>
      <last-name>Doe</last-name>
      <ns1:gender>F</ns1:gender>
   </personal-info>
   <ns1:contact-info>
      <billing-address>
         <street>123 Any Street</street>
         <street>Suite 1</street>
         <city>updatedCity</city>
         <state>ON</state>
         <zip-code>A1B 2C3</zip-code>
      </billing-address>
      <shipping-address>
         <street>456 Another Street</street>
         <city>Raleigh</city>
         <state>NC</state>
         <zip-code>12345</zip-code>
      </shipping-address>
      <ns1:phone-number number-type="cell">(613) 555-2222</ns1:phone-number>
      <ns1:phone-number>12345</ns1:phone-number>
   </ns1:contact-info>
   <changeSummary logging="false" create="#/Customer/ns1:contact-info/ns1:phone-number[2]"
delete="#/Customer/changeSummary/ns1:contact-info/ns1:phone-number[1]"
xmlns:sdo="commonj.sdo">
```

```
      <ns1:contact-info sdo:ref="#/Customer/ns1:contact-info">
         <ns1:phone-number number-type="work">(613) 555-1111</ns1:phone-number>
         <ns1:phone-number sdo:ref="#/Customer/ns1:contact-info/ns1:phone-number[1]"/>
      </ns1:contact-info>
      <billing-address sdo:ref="#/Customer/ns1:contact-info/billing-address">
         <city>Ottawa</city>
      </billing-address>
   </changeSummary>
</Customer>
```

You can see that data graph now includes the change summary. Change summary has enough information to rollback the data graph to the old state.

Concrete Project Using SDO Approach

The one concrete project made for Greenpeace UK last year will be presented to illustrate using of SDO approach.
Project information:

1. Requirements
   Greenpeace Web Portal should provide following functionality:
   - Self Registration (CRUD for visitors and supporters including email confirmation, control of web accounts, managing and validation of personal data)
   - Payment (single donations and recurring payments)
   - E-mail news (notifications about actual Greenpeace events and actions)
   - Campaigning service
2. Duration: 6 month
3. Team: 3 persons
4. Platform, technologies: SOPERA ESB; EclipseLink SDO/JPA; DB2
5. Methodology: SCRUM

The project was implemented using three service layers:

1. Process Layer: primary validation and orchestration of service calls
2. Entity Layer: business logic and persistency
3. Utility Layer: reusable wrapper around external systems: payment, mail, campaigning

At the beginning, the project has faced exactly the problem explained in previous chapters: complex data graph (Greenpeace supporter) is requested by client, modified and sent back to the service for validation, processing and persistency. After some evaluations we have decided to use SDO for communication between client and service façade as well as for communication between different service layers.
It was very easy to validate client changes, apply additional business logic and persist data in consistent way using SDO. Persistency was implemented using SDO-JAXB Bridge provided by EclipseLink team.
The project was successfully finished in terms and budge.

Conclusions and Lessons Learned

1. SDO is a possible solution for stateless SOA Design challenges, especially in case of processing complex Data Graph
2. Change Summary is very comfortable mechanism to validate, process and propagate changes between service consumer and provider
3. It is quite easy to support SDO in ESBs with pluggable marshalling layer (supported by Apache CXF, SOPERA ASF out of the box)

4. Persistency for SDO Graph can be proceeded using JAXB bridge. It allows to use all popular JPA implementations
5. As open source implementations I would recommend to evaluate EclipseLink SDO

Contact address:

Name	**Andrei Shakirin**
Company	Talend GmbH
Address	Banater str, 4a
Postal code, city	85276 Pfaffenhofen a. d. Ilm
Phone:	+49 170 9295908
Fax:	+49(0)12-3456788
Email	ashakirin@talend.com