Service Data Objects White Paper

November 2003
Updated March 2007

Copyright Notice, Trademarks and Terms of Use

Please refer to http://www.osoa.org/display/Main/Terms+and+Conditions

Table of Contents

Introduction

Service Data Objects (SDO) [1] is a specification for a programming model that unifies data programming across data source types, provides robust support for common application patterns, and enables applications, tools, and frameworks to more easily query, view, bind, update, and introspect data. This white paper discusses the motivation behind SDO, presents the SDO architecture and discusses opportunities for further enhancements by vendors in the industry.

Motivation

While the Java™ platform and J2EE provide a variety of data programming models and APIs, these technologies are fragmented and are not always amenable to tooling and frameworks. Further, some of the technologies can be hard to use and may not be sufficiently rich in functionality to support common application needs. SDO is intended to create a uniform data access layer that provides a data access solution for heterogeneous data sources in an easy-to-use manner that is amenable to tooling and frameworks. SDO is not motivated by a need to replace lower-level data access technologies (see Relationship to Other Technologies below). SDO has the following goals:

Unified data access to heterogeneous data sources. Current data programming technologies are more or less specific to data source types. In real-world applications, however, data frequently comes from a variety of sources, including relational databases (accessed through JDBC [2], Entity EJB [3], or other persistence frameworks), custom data access layers (implemented with well-known design patterns), Web services (accessed through JAX-RPC [4] or some other means), XML data stores, JMS [5] messages, and enterprise information systems (accessed through JCA [6] Resource Adapters or via some custom APIs). This heterogeneity poses a significant challenge for application developers because of the broad diversity of programming models they need to learn and use. The plethora of data access APIs is an even more significant challenge for tools and frameworks that attempt to automate many common data programming tasks, such as binding UI components to back-end data sources (e.g., JSR-227 [7]). Thus, a common facility for representing collections of data regardless of data source type can provide a simpler, unified programming model for the application programmer, as well as providing new opportunities for tools and frameworks to work across heterogeneous data sources consistently.

Unified support for both static and dynamic data APIs. Static, strongly typed interfaces provide an easy to use programming model for application programmers. For example, Entity EJBs, JDO [8], Hibernate [9], and other object/relational persistence mechanisms provide typed Java interfaces to data that provides a convenient programming model to developers (e.g., `account.getFirstName()` or `account.getBalance()`). JDBC's ResultSet and RowSet interfaces, in contrast, provide only dynamic, untyped data APIs (e.g., `rs.getString("FIRST_NAME")` or `rs.getFloat("BALANCE")`). Similarly, JAXB [10] provides code-generated Java interfaces to XML data, which makes XML data programming much simpler than using the DOM or SAX APIs.

Neither static nor dynamic data APIs alone are sufficient, however: both are necessary. Static data APIs provide the ease-of-use application programmers needs. But in some cases, static Java interfaces for data is neither feasible nor desirable. For example, dynamic queries where the

shape of the resulting data is not known *a priori*, static Java interfaces for the data is not feasible. Thus, a unified data programming technology needs to support both static and dynamic data APIs seamlessly.

Support for tools and frameworks. Current data programming technologies are not amenable to tools and frameworks across data source types. Tools and frameworks require several key features:

- Uniform access to data across heterogeneous sources. As real-world applications use data from a variety of sources, frameworks need to rely on a uniform data representation.
- Dynamic data APIs. As discussed above, it is much more appropriate for frameworks to use dynamic data APIs, as frameworks are generic to particular data and code-generated interfaces are often inappropriate or not feasible.
- Simple introspection (or metadata) APIs. Frameworks commonly need to introspect the "shape" of data in order to perform data rendering, updateable forms, data binding, etc.

Support for disconnected programming models. Many applications naturally have a disconnected usage pattern of data access: an application reads a set of data, retains it locally for a short period of time, manipulates the data, and then applies the changes back to the data source. For example, this is a very common pattern in Web-based applications: a Web client requests to view a form, a Servlet or JSP requests data in a local read transaction and renders the data in an HTML form, the Web client submits updates to a form, and the Servlet or JSP uses a new transaction to update the data. Best practice typically dictates that optimistic concurrency semantics be used for this scenario, which provides for high levels of concurrency with the appropriate business-level semantic.

There is no data access technology today that provides this disconnected, optimistic model in conjunction with the other features described above. Extensions to JDBC (through the CachedRowSet in JSR-114, "RowSet Implementations" [11]) provide the disconnected model, but as mentioned earlier, JDBC does not meet the requirements of heterogeneous data access or ease-of-use. Similarly, many Object/Relational persistence mechanisms (e.g., many Entity EJB implementations, JDO, Hibernate, etc.) support optimistic concurrency semantics, but they do not provide the necessary universal data access features.

Support for custom data access layers based on common design patterns. Many applications use well-known design patterns (e.g., Transfer Object [12], Transfer Object Assembler [13], Data Access Objects [14], and Domain Store [15]) to build custom data access layers. These patterns are commonly used when the application needs to be insulated from the physical data sources. Implementing data access layers commonly requires a significant amount of custom code, much of which can be automated. Further, these custom data access layers should be able to be integrated into tools and frameworks.

Decouple application code from data access code. In order to be reusable and maintainable, application code should be separable from data access. Data access technologies should be amenable to this separation of concerns.

Architecture

SDO has a composable (as opposed to monolithic) architecture. The core SDO specification provides the base APIs that are applicable to all types of data sources. The core SDO specification, for example, does not presume a particular query language or a particular back-end store. Thus, SQL can be used just as well as XPath [16] or XQuery [17], or any other query language for that matter. Relational databases can be used, as can object databases or XML data sources. The philosophy of the SDO architecture is to use common facilities where possible, and to allow for data-source-type-specific facilities where necessary. The core SDO specification creates the kernel that makes this flexibility and simplicity possible.

The SDO architecture is based upon the concept of *disconnected data graphs*. Under the disconnected data graphs pattern, a client retrieves a data graph (i.e., a collection of tree-structured or graph-structured data) from a data source, mutates the data graph, and can then apply the data graph changes back to the data source. Most commonly, the update is performed with optimistic concurrency semantics, which means that if any of the underlying data was changed before the client applies the changes, the update is rejected and the application must take corrective action. Optimistic concurrency semantics is a natural semantic that serves most business applications well.

Access to data sources is provided by a class of components called *data access services.* A data access service (DAS) is responsible for querying data sources, creating graphs of data containing data objects, and applying changes to data graphs back to data sources. The client is typically disconnected from the data access service: it connects only to retrieve or update a data graph.

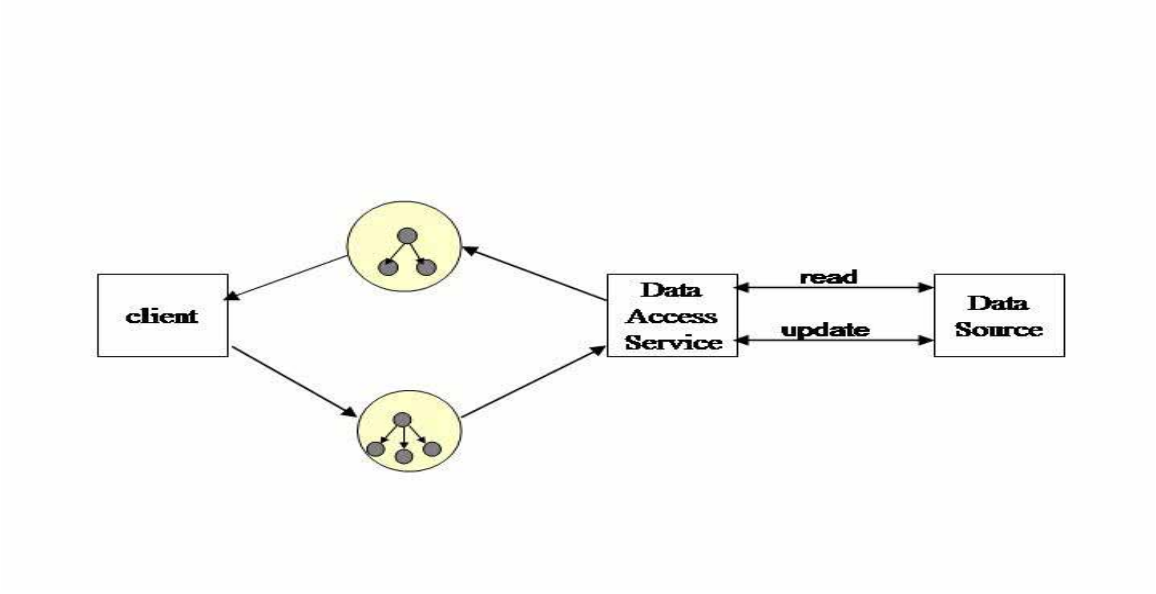

Figure 1: Disconnected data graph architecture

SDO provides a core set of components and services, which are then augmented by SDO-enabled tools and frameworks. The SDO architecture includes the following:

- **SDO Core.** The core SDO specification [1] contains the primary components programmers routinely work with, including Data Objects, which represents the data, and Data Graphs, which represents the data graph. The SDO specification also provides a metadata API that allows clients to introspect the data model. The SDO metadata API enables the tools and frameworks to work uniformly with heterogeneous data sources. The metadata API supports essential concepts from other more comprehensive metamodels, such as W3C XML Schema [18], the SQL relational model, and the Essential Meta Object Facility (EMOF) [19].

- **SDO Data Access Services.** A Data Access Service provides access to data sources. Data Access Services create Data Graphs by reading data from backend data sources, and can update data sources based on changes made to data graphs. Data Access Services can come in various shapes and sizes, and could include, for example, XML Data Access Service that can read and write to XML data sources, relational Data Access Service that can read and write to JDBC-based data sources, application-defined entity models, or Data Access Services that accept XML Query and read and write a variety of XML and non-XML data sources. Specifications for Data Access Services are on the SDO roadmap.

- **SDO-enabled Tools.** SDO-enabled tools include code generators, metamodel converters, schema converters, data modeling tools, schema modeling tools, etc.

- **SDO-enabled Runtimes and Frameworks.** The runtimes and frameworks work with the various components in SDO to perform a variety of tasks, such as data binding to UI components. The SDO architecture is a key enabler for these runtimes and frameworks.

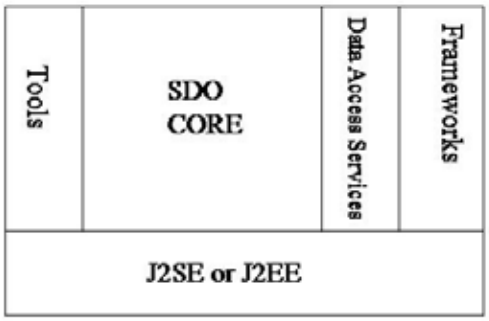

Figure 2: SDO and the Java platform

The various components in this architecture are Data Objects, Data Graphs, and Data Access Services. Additionally, metadata plays a key role. These components have the following responsibilities:

- **Data Object.** Data Objects hold the actual data, including primitive values and references to other Data Objects. Data Objects also have references to their metadata, which allows the Data Object to be introspected for information about data types, data relations, and data constraints.

- **Data Graph.** A Data Graph is conceptually a set of data that provides the unit of transfer between components or tiers. In particular, a data draph is a multi-rooted set of Data Objects. The data graph records all changes to data, including new Data Objects, changed Data Objects, and removed Data Objects.

- **Metadata.** Metadata about Data Objects enables development tools and runtime frameworks to introspect data, including data types, relationships, and constraints. SDO provides a common metadata API across data source types to enable generic tools and frameworks.

- **Data Access Service.** A Data Access Service(DAS) is responsible for interacting with a data source to produce Data Graphs representing the data. The Data Access Service is also responsible for applying changes in a Data Graph back to a data source.

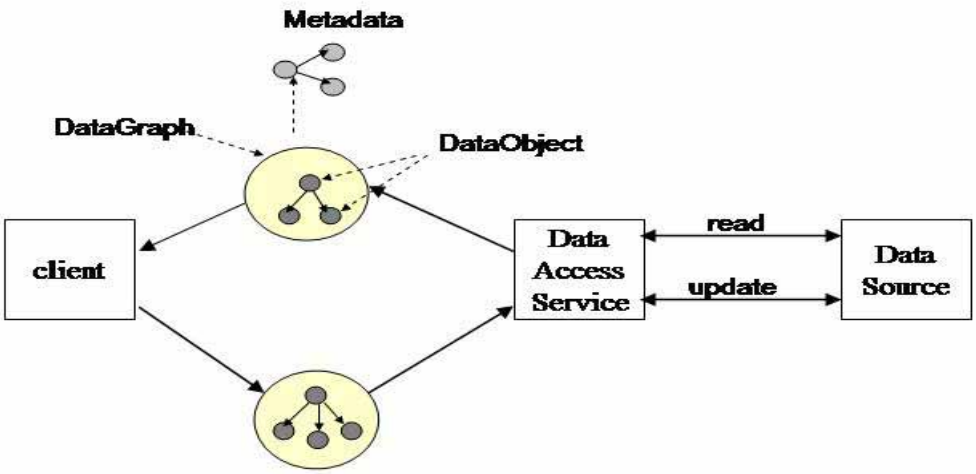

Figure 3: Components of the SDO solution

The relationships between the objects in SDO can be expressed in a UML model:

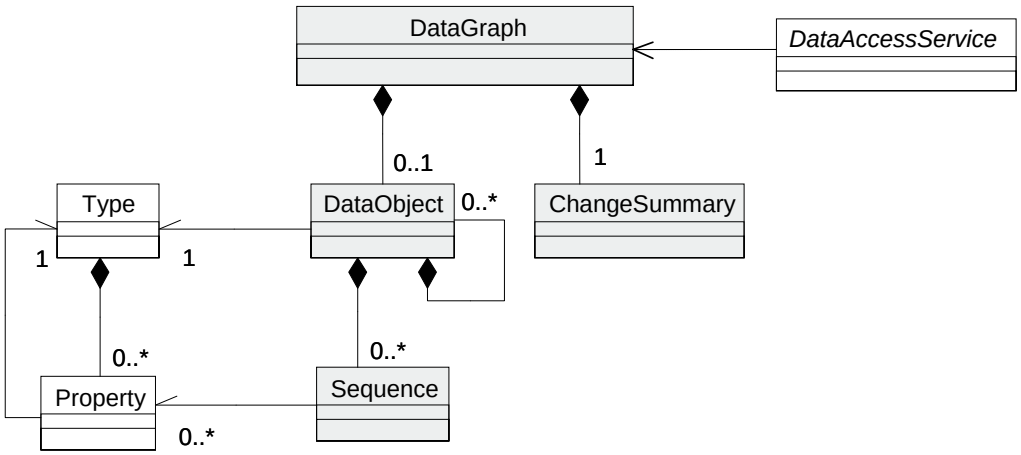

Figure 4: UML model of core SDO components

Data Objects

A Data Object holds data as a set of *properties.* These properties include both primitive values (such as XML attributes and XML element contents) and references to other Data Objects (e.g., a PurchaseOrder Data Object might have a Customer property). In the case of XML data, a Data Object would commonly represent an element, the element's attributes, values of simple-type child elements, and references to Data Objects representing complex-type child elements. In the case of relational data, a Data Object would commonly represent a row of data. A foreign key would be represented by a reference to another Data Object, representing a row in another table.

Data Objects can be introspected with the SDO metadata API (described below). This allows programs to get information about types, relationships, and constraints. Note that introspection is not necessarily limited to the SDO metadata API: some implementations might provide access to native metadata, such as an XML Schema object model. However, the SDO metadata API provides the most commonly needed metadata and applications should revert to other metamodels only when necessary.

Data Objects offer, at minimum, a dynamic data API for reading and modifying objects, including the object's properties. This dynamic API uses simple XPath expressions to locate Data Objects within the Data Graph. Optionally, static Java interfaces for Data Objects can be generated from models or schemas (e.g., SQL relational schemas, XML Schema definitions, EMOF models, etc.). This provides a user-friendly API at development time. For example, given an XML Schema definition for a Purchase Order, a PurchaseOrder Java interface can be generated. Similarly, given JDBC metadata for a PURCHASE_ORDER table, a PurchaseOrder Java interface can be generated.
The core SDO specification does not define Java interface generation for Data Objects. Existing Java interface generation solutions and specifications can be used and integrated with SDO (e.g., JAXB). For example, many solutions exist for generating Java interfaces from XML Schema definitions. Java interface generation for SQL relational data and data from other sources is also possible.

Note that static Java interfaces for Data Objects are not always necessary or desirable. The dynamic data API is important because code generation is not appropriate for all circumstances. There will be many cases where runtime frameworks will interpret metadata and use the dynamic data API. However, statically typed interfaces provide ease-of-use to the programmer at development time (e.g., compile-time type checking, availability of code completion, etc.).

Data Objects have rich relationship support, including support for 1:1, 1:n, and n:m relationships. Data Objects manage relationships, including relationship inverses. For example, say Data Object A references Data Object B, and B maintains an inverse reference to A. Then, the program moves B from A to some Data Object C. In this scenario, B's relationship will automatically be changed from A to C. Containment (or ownership) relationships are also supported, which enables Data Access Sevices to make proper decisions (e.g., cascading deletes).

Data Graph

A Data Graph represents a set of data. More specifically, it holds a set of Data Objects. Data Graphs are typically the unit of transfer between components in a system. As such, Data Graphs are responsible for recording change summaries. Data Graphs can optionally track change history for all the Data Objects in the Data Graph. This change history may be accessed in the form of a Change Summary, typically by a data access service service that wishes to apply the changes to a back-end data source.
The Change Summary provides information on which Data Objects have been added, removed, and updated. For updated Data Objects, the Change Summary provides both old and new values for updated properties.

Metadata

Data is sometimes described in terms of layers of instance data, metadata, and metamodel. The base layer is the data itself, or the instance data. This data conforms to a data model or schema, which is described by the metadata. This metadata itself conforms to a model, which is called the *metamodel.*

To draw an analogy to XML, an XML instance document is the instance data, the schema (e.g. an XML Schema Definition, or XSD) is the metadata, and the schema language (e.g. W3C XML Schema) is the metamodel. Similarly, with relational data, the ResultSet or RowSet contains the data, the data definition is the metadata (surfaced through the ResultSetMetaData interface), and the SQL relational model is the metamodel.

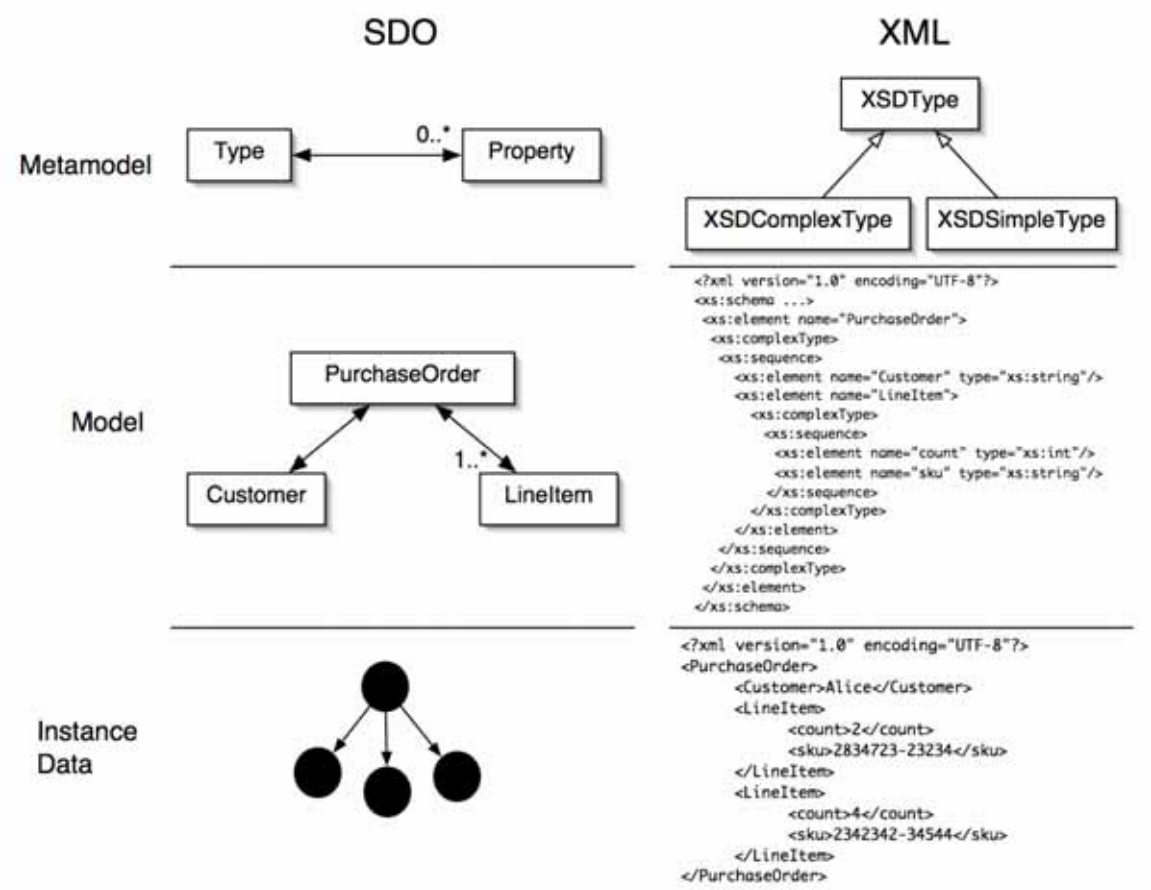

Figure 5: Relationship between data, metadata, and metamodels

SDO provides a small metadata API (and implicitly, a slim metamodel). This metamodel is not meant to be comprehensive like XML Schema, the SQL relational metamodel, or EMOF. Rather, it provides the essential client view of the metadata that applications and frameworks need for common introspection tasks.

This enables a variety of use cases where normalized introspection of data is useful. A common use case is tools that perform data binding between Web UI components and data sources. SDO

enables these data binding frameworks to work with XML data, relational data, JCA record data, etc. independent of their native metamodels.

Use Cases

SDO enables many common use cases, a few of which are described here.

Virtual Data Access

SDO can be used in a virtual data access architecture that provides universal, aggregated data from multiple, heterogeneous data sources. This is a common architectural pattern, even when the underlying data sources are all relational data accessed through JDBC or entity EJBs. A "virtual data access service" could support rich query languages, such as XQuery. The data graph from the virtual data access service can then be inspected, introspected, updated, and then sent back to the original data sources – all without the SDO programmer needing to know or learn intricacies of back-end data access across various types. This is an extremely critical requirement in enterprise environments where "data provision" function is formally separated from "data use" function for variety of reasons including skills, security, and governance. Further, this separation is also very important in Service-Oriented Architecture (SOA) implementations. SDO provides the unified client data and metadata API that enables tools, frameworks, and runtimes to function in this environment.

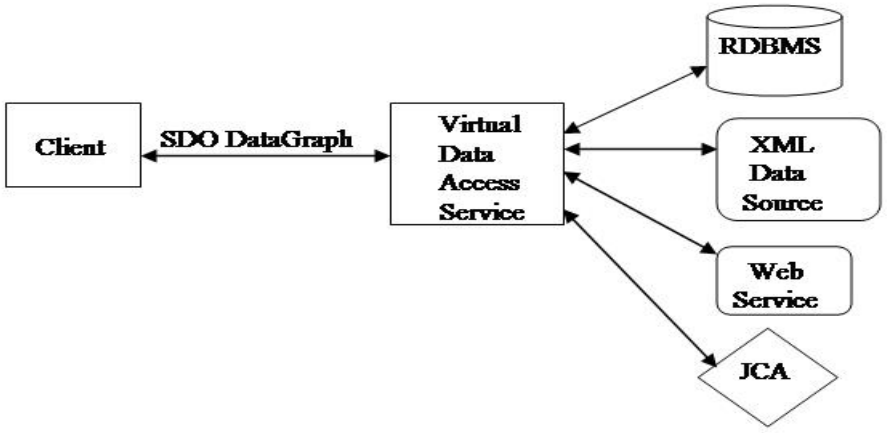

Figure 6: Using SDO for virtual data access

Relational Database Access

The SDO programming model is compelling for programming against relational data sources because it provides the disconnected access model along with both static and dynamic data APIs. In fact, the SDO architecture is essentially a codification of widely used J2EE design patterns,

including Transfer Object, Transfer Object Assembler, and Domain Store. A relational data access service can provide SQL-based query capabilities. The relational data access could use JDBC to implement queries and updates:

Figure 7: Using SDO for relational data

SDO can also integrate with object-relational persistence mechanisms, including Entity EJB, JDO, and Hibernate.

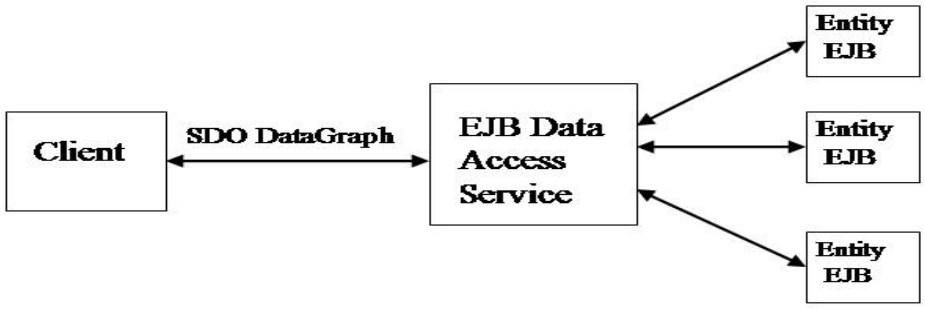

Figure 8: Using SDO with Entity EJBs

Reading and Writing XML

SDO enables a comprehensive programming model for XML, including querying, reading, and updating. Under this architecture, the XML data source could be an XML file, a native XML data store, or a relational database with XML capabilities. Note that SDO is meant to provide a higher-level abstraction than XQuery APIs, such as JSR-225 ("XQuery API for Java") [20]. For example, an "XML Data Access Service" that can talk to an XQuery-capable XML data source might works as follows:

Figure 9: Using SDO in an XML use case

Opportunities

This SDO specification provides the core of the customer solution. We anticipate that many vendors will build tools, data access services and frameworks which will further integrate SDO with existing and future data access technologies.

Tool support. SDO enables a wide range of tool support. Java interface generation tools are necessary to provide static data APIs that integrate with SDO. For example, a tool could be provided that generates static data APIs from XML Schema and is integrated with the SDO APIs. Likewise, a tool could be provided that generates static data APIs from relational schemas and is integrated with the SDO APIs. Yet another tool could be used to generate interfaces from UML [21] models. In fact, many of these tools exist today, and simply need to be extended to support SDO. Follow-on specifications may define how Java interfaces are generated from various data sources if there is not an appropriate specification already.

Data Access services. SDO requires data access services to be useful. Initially, vendor-specific data access services will exist. Existing persistence mechanisms, such as popular object/relational persistence frameworks, could be extended to become SDO-capable data access services. Similarly, XQuery engines could be turned into SDO-capable data access services. As more experienced is gained building data access services, it will be useful to standardize data access service interfaces.

Frameworks. SDO enables a variety of frameworks that will provide ease-of-use for many common data programming tasks. For example, frameworks could use SDO to automate custom data access layers that use the Transfer Object, Transfer Object Assembler, Data Access Object, and Domain Store design patterns. Other frameworks could use SDO to bind user interface components (e.g., grids, list boxes, etc.) to back-end data sources. JSR-227 proposes such a framework, and it could use SDO to provide the necessary primitives.

Relationship with Other Technologies

SDO integrates seamlessly other data programming technologies. We describe here how SDO relates to, and integrates with, other data programming technologies and APIs.

The following table compares SDO to other data programming technologies:

	Model	API	Data Source	Metadata API	Query Language
JDBC RowSet	Connected	Dynamic	Relational	Relational	SQL
JDBC CachedRowSet	Disconnected	Dynamic	Relational	Relational	SQL
JPA, Entity EJB	Connected	Static	Relational	Java introspection	EJBQL
JDO	Connected	Static	Relational, Object	Java introspection	JDOQL
JCA	Disconnected	Dynamic	Record-based	Undefined	Undefined
DOM and SAX	N/A	Dynamic	XML	XML infoset	XPath, XQuery
JAXB	N/A	Static	XML	Java introspection	N/A
JAX-RPC	N/A	Static	XML	Java introspection	N/A
SDO	Disconnected	Both	Any	SDO metadata API, Java introspection	Any

JDBC and JSR-114

JDBC provides a connected model for programming against relational data sources with a dynamic data API. JSR-114 ("RowSet Implementations") provides extensions to the JDBC that provides some, but far from all, of the features enabled by SDO. JSR-114 provides the JoinRowSet, which can represent data across table joins, and the CachedRowSet, which provides a disconnected model for data access.

SDO provides both of these features, but differs from JDBC and JSR-114 in two key ways: (a) JDBC is meant for data access to relational data sources, while SDO can be used with any type

of data source, and (b) JDBC provides only a dynamic data API, while SDO naturally supports static data APIs and dynamic data APIs.

A data access service for relational data would presumably use JDBC and possibly JSR-114 for its implementation.

JPA and Entity EJB

SDO can be used both in conjunction with JPA and Entity EJBs via a data access service that interfaces with the entity beans. In fact, this architecture is a manifestation of the commonly used Transfer Object and Transfer Object Assembler design patterns.
In some cases, using SDO in conjunction with a relational data access service can obviate the need for Entity EJB. However, Entity EJB provides a connected, transactional model that is more appropriate for some applications. SDO could be used with a JPA implementation as well.

JDO, Hibernate, and Other Persistence Frameworks

JDO, Hibernate, and other object-relational persistence frameworks provide a variety of the characteristics that SDO provides. For example, both JDO and Hibernate provide the convenient static data APIs. Hibernate, and some JDO implementations, provide the optimistic concurrency, disconnected model. These persistence frameworks could be extended to become SDO-capable data access services, which allows these frameworks to work within the SDO solution.

Java-XML Binding

Java-XML binding frameworks, such as JAX-RPC, JAXB, SAAJ [22], XML Beans [23], and the Eclipse EMF [24] and XSD [25] tools enable XML data to be bound to Java objects. These Java interfaces providing access to the data are generated from schema languages such as XML Schema or RELAX NG [26]. These generated interfaces provide a programmer-friendly way of reading, writing, and manipulating XML data.

Java-XML binding frameworks can be extended such that they are integrated with SDO. For example, JAXB could be used as the tool that provides the static data API, and be extended to support the Data Graph and Data Object interfaces.

XQuery

XQuery is the forthcoming standard for querying XML data and can be used with XML data sources. XQuery can also be used more generally against any data repository. Thus, a data access service can use XQuery for the query language.

Web UI Data Binding and JSR-227

JSR-227 (“A Standard Data Binding & Data Access Facility for J2EE”) proposes a facility to declaratively bind Web user interface components to heterogeneous back-end data sources. For example, JavaServer Faces components (e.g., a grid component) could be bound to relational data or XML data from a Web service. The SDO specification provides the lower-level primitives that could be used with JSR-227.

JCA and JMS

The JCA Common Client Interface (CCI) uses the Record abstraction to communicate data with enterprise resources. These records have no particular structure. With SDO, the client application could use Data Graphs to communicate with the enterprise resources. Similarly, JMS has an unstructured Message construct. SDO Data Graphs can be used to send and receive data on JMS queues and topics.

References

[1] Service Data Objects Specifications,
http://www.osoa.org/display/Main/Service+Data+Objects+Specifications
[2] JDBC Data Access API,
http://java.sun.com/products/jdbc/
[3] Enterprise JavaBeans,
http://java.sun.com/products/ejb/
[4] Java API for XML-Based RPC,
http://java.sun.com/xml/jaxrpc/
[5] Java Message Service,
http://java.sun.com/products/jms/
[6] J2EE Connector Architecture,
http://java.sun.com/j2ee/connector/
[7] A Standard Data Binding & Data Access Facility for J2EE,
http://www.jcp.org/en/jsr/detail?id=227
[8] Java Data Objects,
http://java.sun.com/products/jdo/
[9] Hibernate,
http://www.hibernate.org/
[10] Java API for XML Binding,
http://java.sun.com/xml/jaxb/
[11] JDBC RowSet Implementations,
http://jcp.org/en/jsr/detail?id=114
[12] Core J2EE Patterns - Transfer Object,
http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html
[13] Core J2EE Patterns - Transfer Object Assembler,
http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObjectAssembler.html
[14] Core J2EE Patterns – Data Access Object,
http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
[15] Core J2EE Patterns: Best Practices and Design Strategies, 2nd ed., Deepak Alur, John Crupi and Dan Malks
[16] XPath,
http://www.w3.org/TR/xpath
[17] XQuery,
http://www.w3.org/TR/xquery/
[18] XML Schema,
http://www.w3.org/TR/xmlschema-1/ and http://www.w3.org/TR/xmlschema-2/
[19] Meta Object Facility,
http://www.omg.org/docs/ad/03-04-07.pdf

[20] XQuery API for Java,
http://www.jcp.org/en/jsr/detail?id=225
[21] Unified Modeling Language,
http://www.omg.org/uml/
[22] SOAP with Attachments API for Java,
http://java.sun.com/xml/saaj/
[23] XMLBeans,
http://xml.apache.org/xmlbeans/
[24] Eclipse Modeling Framework,
http://www.eclipse.org/emf/
[25] Eclipse XML Schema Infoset Model,
http://www.eclipse.org/xsd/
[26] RELAX NG Specification,
http://www.oasis-open.org/committees/relax-ng/spec-20011203.html