Chapter No. 4
"Data and Services—All Roads Lead to Enterprise Service Bus"
In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.4 "Data and Services—All Roads Lead to Enterprise Service Bus"

A synopsis of the book's content

Information on where to buy this book

About the Authors

Malhar Barai is a senior systems analyst with Satyam Computer Services Ltd., one of India's leading IT services organizations. He has more than seven years of experience in the industry working for leading organizations across India.

Malhar has interest in service-oriented technologies and application integration tools. He has worked on EAI toolset of webMethods and Cast Iron, Java technologies.

You can catch him on various forums that deal with SOA and some of the webMethods forums, or you can read about him on his blog http://malharbarai.blogspot.com

He gets spurred by the daily challenges at work, finding solutions to the problems, and trying his hand at improving processes and solutions.

I would like to acknowledge and dedicate this book to my parents for being sources of inspiration and for guiding me on the right path when it mattered the most. To Jalpa, my lovely wife for, being a constant support and carving out a wonderful life for us. My ex-manager Ajay Mulkalwar for his guidance and encouragement, and the most important person—my soul, my sweet daughter Preisha whose lovely smile makes my time wonderful…

For More Information:
www.packtpub.com/service-oriented-architecture-for-java-applications/book
Vincenzo Caselli graduated with a degree in electrical engineering in 1991 from the University of Bologna. He has worked as an independent consultant and a Java trainer for several Italian software houses since 1996. He began working as a developer in Delphi and other visual IDE’s with AS/400-based companies. Soon he shifted his focus on Java and began to propose Swing client/server multi-layered solutions to his customers. He also worked in the web development area with several frameworks (Struts, Hibernate, Spring, JSF, and GWT) in different fields (banking, manufacturing, healthcare, e-learning). Recently, he collaborated with IBM in projects based on Eclipse RCP and SOA. He is interested in consultancy and training activities aimed to improve the productivity and quality of the software development process by using open-source products.

I would like to thank my wife Silvia and my daughter Linda for being patient while I worked on this book. I also want to thank my friend Luca Masini for his precious technical advice and help.

Binildas C. A. provides Technical Architecture consultancy for IT solutions. He has more than 13 years of IT experience, mostly in Microsoft and Sun technologies. Distributed Computing and Service Oriented Integration are his mainstream skills, with extensive hands-on experience in Java and C#.NET programming. Binil holds a Bachelor of Technology degree in mechanical engineering from the College of Engineering, Trivandrum (www.cet.ac.in) and an MBA in systems management from Institute of Management, Kerala (www.imk.ac.in). A well-known and a highly sought-after thought leader, Binil has designed and built many highly scalable middle-tier and integration solutions for several top-notch clients including Fortune 500 companies. He has been previously employed by multiple IT consulting firms including IBS Software Services (www.ibsplc.com) and Tata Consultancy Services (www.tcs.com), and he currently works for Infosys Technologies (www.infosys.com) as a Principal Architect where he heads the J2EE Architects group servicing Communications Service Provider clients.

Binil is a Sun Certified Programmer (SCJP), Developer (SCJD), Business Component Developer (SCBCD) and Enterprise Architect (SCEA), Microsoft Certified Professional (MCP), and Open Group (TOGAF8) Certified Enterprise Architecture Practitioner. He is also a Licensed Zapthink Architect (LZA) in SOA. Besides Technical Architecture, Binil also practices Enterprise Architecture.

When not in software, Binil spends time with wife Sowmya and daughter Ann in 'God's Own Country', Kerala (www.en.wikipedia.org/wiki/Kerala). Binil is a long distance runner and is a national medalist in power lifting. You may contact Binil at biniljava@yahoo.co.in or binil_christudas@infosys.com.
Service Oriented Architecture with Java

Service Oriented Architecture is mainly a mindset, an enterprise strategy whose natural implementation is represented by web services. SOA is not a single product or single reference architecture to be followed, but SOA is all about best practices, reference architectures, processes, toolsets, and frameworks, along with many other things which will help you and your organization to increase the responsiveness and agility of your enterprise architecture. Standards and frameworks play a greater role in enabling easy and widespread industry adoption of SOA.

This book will help you learn the importance of designing a sound architecture for successful implementation of any business solution, different types of C/S architecture, and various tenets of SOA, explaining the fundamentals and explaining the advantage of using the Service Oriented Architecture in designing of the business solution. From a basic XML-over-HTTP approach to the REST and SOAP protocols, we get into the details of how web services can be implemented with various degrees of complexity and flexibility using JAVA.

This book will explain the concepts of business layer that is 'The SOA core'. You will also learn when SOA will define as an asset to your project with the help of practical examples.

In the early years when the WS-approach began to emerge it suffered from difficulties due to many factors, for instance, complex adoption process and poor standardization. Now, with little effort times are mature for using this technology and also getting great advantages, both immediate and as an investment for our future works. The book concludes with the focus on explanation of these assets.
What This Book Covers

In Chapter 1 we will discuss the role of Architecture for successful implementation of any business solution followed by brief discussion on different types of client-server architecture and SOA.

In Chapter 2 we will examine the relationship between the SOA methodology and the web service implementation basics. We will also discuss how XML can be used as the common language to decouple the communication between web service implementations and their consumer clients.

In Chapter 3 we will introduce major web service implementations available specifically in the Java and J2EE world, WS using JAX-WS 2.0, WS using Apache Axis, WS using Spring, and WS using XFire.

In Chapter 4 we shall see few emerging standards like SDO and SCA, addressing from data integration to service and component integration.

In Chapter 5 we will look into a couple of case studies where one of the solutions is based on principles of Enterprise Application Integration and in the second one we shall build our solution based on SOA fundamentals.

In Chapter 6 we will explore in detail the advantages that the SOA approach can lead to. Basically a concluding chapter discussing what we can and what we have achieved with SOA approach.
Data and Services—All Roads Lead to Enterprise Service Bus

Having seen the basics of XML and XML-based services in the previous chapters, we are now ready to look into the big picture of enterprise landscape and see how all the pieces fit together. What is of interest for every enterprise user is information and every information starts from the basic building block, data. Data can reside in any data store, and can exist in many formats. Irrespective of that, you need to bring data to your table, do some massaging with your business use cases, and supply them as information. How do we do that in the SOA world, moving away from the traditional JDBC or Object-relational mapping (OR mapping) styles? And more interesting is, data can even exist in the form of services and if so, how do we combine multiple services just like we combine data from multiple JDBC query results? We are going to look at a couple of these aspects in this chapter, and we will cover specifically:

- JDO as an alternative to JDBC
- Data Services and its role in SOA
- Few emerging Data Services standards like SCA and SDO
- Introducing Apache Tuscany
- Introduction to message-oriented middleware (MOM)
- Enterprise Service Bus (ESB)—The new architecture style
- Introducing OpenESB
You all are perfectly comfortable with JDBC or few OR-mapping frameworks at least, like Hibernate or TopLink. Let us now look into a complementing standard of accessing data from your data store using a standard interface-based abstraction model of persistence in java that is, **Java Data Objects (JDO)**. The original JDO (JDO 1.0) specification is quite old and is based on **Java Specification Request 12 (JSR 12)**. The current major version of JDO (JDO 2.0) is based on JSR 243. The original specifications were done under the supervision of Sun and starting from 2.0, the development of the API and the reference implementation happens as an Apache open-source project.

**Why JDO?**

We have been happily programming to retrieve data from relational stores using JDBC, and now the big question is do we need yet another standard, JDO? If you think that as software programmers you need to provide solutions to your business problems, it makes sense for you to start with the business use cases and then do a business analysis at the end of which you will come out with a **Business Domain Object Model (BDOM)**. The BDOM will drive the design of your entity classes, which are to be persisted to a suitable data store. Once you design your entity classes and their relationship, the next question is should you be writing code to create tables, and persist or query data from these tables (or data stores, if there are no tables). I would like to answer 'No' for this question, since the more code you write, the more are the chances of making errors, and further, developer time is costly. Moreover, today you may write JDBC for doing the above mentioned "technical functionalities", and tomorrow you may want to change all your JDBC to some other standard since you want to port your data from a relational store to a different persistence mechanism. To sum up, let us list down a few of the features of JDO which distinguishes itself from other similar frameworks.

- **Separation of Concerns**: Application developers can focus on the BDOM and leave the persistence details (storage and retrieval) to the JDO implementation.

- **API-based**: JDO is based on a java interface-based programming model. Hence all persistence behavior including most commonly used features of OR mapping is available as metadata, external to your BDOM source code. We can also Plug and Play (PnP) multiple JDO implementations, which know how to interact well with the underlying data store.
Chapter 4

- **Data store portability**: Irrespective of whether the persistent store is a relational or object-based file, or just an XML DB or a flat file, JDO implementations can still support the code. Hence, JDO applications are independent of the underlying database.

- **Performance**: A specific JDO implementation knows how to interact better with its specific data store, which will improve performance as compared to developer written code.

- **J2EE integration**: JDO applications can take advantage of J2EE features like EJB and thus the enterprise features such as remote message processing, automatic distributed transaction coordination, security, and so on.

**JPOX—Java Persistent Objects**

JPOX is an Apache open-source project, which aims at a heterogeneous persistence solution for Java using JDO. By heterogeneous we mean, JPOX JDO will support any combination of the following four main aspects of persistence:

- **Persistence Definition**: The mechanism of defining how your BDOM classes are to be persisted to the data store.
- **Persistence API**: The programming API used to persist your BDOM objects.
- **Query Language**: The language used to find objects due to certain criteria.
- **Data store**: The underlying persistent store you are persisting your objects to.


**JDO Sample Using JPOX**

In this sample, we will take the familiar Order and LineItems scenario, and expand it to have a JDO implementation. It is assumed that you have already downloaded and extracted the JPOX libraries to your local hard drive.
BDOM for the Sample

We will limit our BDOM for the sample discussion to just two entity classes, that is, OrderList and LineItem. The class attributes and relationships are shown in the following screenshot:

The BDOM illustrates that an Order can contain multiple line items. Conversely, each line item is related to one and only one Order.

Code BDOM Entities for JDO

The BDOM classes are simple entity classes with getter and setter methods for each attribute. These classes are then required to be wired for JDO persistence capability in a JDO specific configuration file, which is completely external to the core entity classes.

OrderList.java

OrderList is the class representing the Order, and is having a primary key attribute that is number.

```java
public class OrderList{
    private int number;
    private Date orderDate;
    private Set lineItems;
    // other getter & setter methods go here
    // Inner class for composite PK
    public static class Oid implements Serializable{
        public int number;
        public Oid() {
        }
        public Oid(int param){
            this.number = param;
        }
        public String toString(){
```
return String.valueOf(number);
}
public int hashCode(){
    return number;
}
public boolean equals(Object other){
    if (other != null && (other instanceof Oid)){
        Oid k = (Oid)other;
        return k.number == this.number;
    }
    return false;
}
}

LineItem java

LineItem represents each item container in the Order. We don’t explicitly define a primary key for LineItem even though JDO will have its own mechanism to do that.

public class LineItem{
    private String productId;
    private int numberOfItems;
    private OrderList orderList;
    // other getter & setter methods go here
}

package.jdo

JDO requires an XML configuration file, which defines the fields that are to be persisted and to what JDBC or JDO wrapper constructs should be mapped to. For this, we can create an XML file called package.jdo with the following content and put it in the same directory where we have the entities.

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd">
<jdo>
    <package name="com.binildas.jdo.jpox.order">
        <class name="OrderList" identity-type="application"
            objectid-class="OrderList$Oid" table="ORDERLIST">
            <field name="number" primary-key="true">
                <column name="ORDERLIST_ID"/>
            </field>
            <field name="orderDate">
                <column name="ORDERLIST_ID"/>
            </field>
        </class>
    </package>
</jdo>
In this sample, we will persist our entities to a relational database, Oracle. We specify the main connection parameters in `jpox.PROPERTIES` file.

```java
javax.jdo.PersistenceManagerFactoryClass=org.jpox.jdo.JDOPersistenceManagerFactory
javax.jdo.option.ConnectionDriverName=oracle.jdbc.driver.OracleDriver
javax.jdo.option.ConnectionURL=jdbc:oracle:thin:@127.0.0.1:1521:orcl
javax.jdo.option.ConnectionUserName=scott
javax.jdo.option.ConnectionPassword=tiger
org.jpox.autoCreateSchema=true
org.jpox.validateTables=false
org.jpox.validateConstraints=false
```

**jpox.PROPERTIES**

Data and Services – All Roads Lead to Enterprise Service Bus

```xml
<class name="ORDER" table="ORDER">
  <field name="ORDER_DATE" />
</class>

<class name="LineItem" table="LINEITEM">
  <field name="productId">
    <column name="PRODUCT_ID" />
  </field>
  <field name="numberOfItems">
    <column name="NUMBER_OF_ITEMS" />
  </field>
  <field name="orderList" persistence-modifier="persistent">
    <column name="LINEITEM_ORDERLIST_ID" />
  </field>
</class>
</package>
</jdo>

For More Information:
www.packtpub.com/service-oriented-architecture-for-java-applications/book
Main.java

This class contains the code to test the JDO functionalities. As shown here, it creates two Orders and adds few line items to each order. First it persists these entities and then queries back these entities using the id.

```java
public class Main{
    static public void main(String[] args){
        Properties props = new Properties();
        try{
            props.load(new FileInputStream("jpox.properties"));
        }
        catch (Exception e){
            e.printStackTrace();
        }
        PersistenceManagerFactory pmf =
                JDOHelper.getPersistenceManagerFactory(props);
        PersistenceManager pm = pmf.getPersistenceManager();
        Transaction tx = pm.currentTransaction();
        Object id = null;
        try{
            tx.begin();
            LineItem lineItem1 = new LineItem("CD011", 1);
            LineItem lineItem2 = new LineItem("CD022", 2);
            OrderList orderList = new OrderList(1, new Date());
            orderList.getLineItems().add(lineItem1);
            orderList.getLineItems().add(lineItem2);
            LineItem lineItem3 = new LineItem("CD033", 3);
            LineItem lineItem4 = new LineItem("CD044", 4);
            OrderList orderList2 = new OrderList(2, new Date());
            orderList2.getLineItems().add(lineItem3);
            orderList2.getLineItems().add(lineItem4);
            pm.makePersistent(orderList);
            id = pm.getObjectId(orderList);
            System.out.println("Persisted id : "+ id);
            pm.makePersistent(orderList2);
            id = pm.getObjectId(orderList2);
            System.out.println("Persisted id : "+ id);
            orderList = (OrderList) pm.getObjectById(id);
            System.out.println("Retrieved orderList : " + orderList);
            tx.commit();
        }
    }
}
```
catch (Exception e) {
    e.printStackTrace();
    if (tx.isActive()) {
        tx.rollback();
    }
}
finally {
    pm.close();
}
}

Build and Run the JDO Sample

As a first step, if you haven’t done it before, edit examples.PROPERTIES provided along with the code download for this chapter and change the paths there to match your development environment. The code download for this chapter also includes a README.txt file, which gives detailed steps to build and run the samples.

Since we use Oracle to persist entities, we need the following two libraries in the classpath:

- jpx-rdbms*.jar
- classes12.jar

We require a couple of other libraries too which are specified in the build.xml file. Download these libraries and change the path in examples.PROPERTIES accordingly.

To build the sample, first bring up your database server. Then to build the sample in a single command, it is easy for you to go to ch04\jdo folder and execute the following command.

    cd ch04\jdo
    ant

The above command will execute the following steps:

- First it compiles the java source files
- Then for every class you persist, use JPOX libraries to enhance the byte code.
- As the last step, we create the required schema in the data store.
To run the sample, execute:

    ant run

You can now cross check whether the entities are persisted to your data store. This is as shown in the following screenshot where you can see that each line item is related to the parent order by the foreign key.
Data Services

Good that you now know how to manage the basic data operations in a generic way using JDO and other techniques. By now, you also have good hands-on experience in defining and deploying web services. We all appreciate that web services are functionalities exposed in standard, platform, and technology neutral way. When we say functionality we mean the business use cases translated in the form of useful information. Information is always processed out of data. So, once we retrieve data, we need to process it to translate them into information.

When we define SOA strategies at an enterprise level, we deal with multiple Line of Business (LOB) systems; some of them will be dealing with the same kind of business entity. For example, a customer entity is required for a CRM system as well as for a sales or marketing system. This necessitates a Common Data Model (CDM), which is often referred to as the Canonical Data Model or Information Model. In such a model, you will often have entities that represent "domain" concepts, for example, customer, account, address, order, and so on. So, multiple LOB systems will make use of these domain entities in different ways, seeking different information-based on the business context. OK, now we are in a position to introduce the next concept in SOA, which is "Data Services".

Data Services are specialization of web services which are data and information oriented. They need to manage the traditional CRUD (Create, Read, Update, and Delete) operations as well as a few other data functionalities such as search and information modeling. The Create operation will give you back a unique ID whereas Read, Update, and Delete operations are performed on a specific unique ID. Search will usually be done with some form of search criteria and information modeling, or retrieval happens when we pull useful information out of the CDM, for example, retrieving the address for a customer.

The next important thing is that no assumptions should be made that the data will be in a java resultset form or in a collection of transfer object form. Instead, you are now dealing with data in SOA context and it makes sense to visualize data in XML format. Hence, XML Schema Definition (XSDs) can be used to define the format of your requests and responses for each of these canonical data definitions. You may also want to use ad hoc queries using XQuery or XPath expressions, similar to SQL capabilities on relational data. In other words, your data retrieval and data recreation for information processing at your middle tier should support XML tools and mechanisms, and should also support the above six basic data operations. If so, higher level of abstractions in the processing tier can make use of the above data services to provide Application Specialization capabilities, specialized for the LOB systems. To make the concept clear, let us assume that we need to get the order status for a particular customer (getCustomerOrderStatus()) which will take the customer ID argument. The data services layer will have a retrieve operation.

For More Information:
www.packtpub.com/service-oriented-architecture-for-java-applications/book
passing the customer ID and the XQuery or the XPath statement will obtain the requested order information from the retrieved customer data. High level processing layers (such as LOB service tiers) can use high-level interface (for example, our `getCustomerOrderStatus` operation) of the Application Specialization using a web services (data services) interface and need not know or use XQuery or XPath directly. The underlying XQuery or XPath can be encapsulated, reused, and optimized.

**Service Data Objects**
Data abstraction and unified data access are the two main concerns that any SOA-based architecture has to address. In the data services discussion, we talked a bit about data abstraction, by first defining data around domain entities and then decorating it with useful methods for data operations. Equally important is the issue of accessing heterogeneous data in a uniform way.

**Why SDO?**
One of the main problems Service Data Objects (SDO) tries to solve is the issue of heterogeneous manner of data management. By data management, we mean data storage as well as operations on data lifecycle. SDO simplifies J2EE data programming model thus giving application developers more time to focus on the business problems.

SDO provides developers an API, the SDO API, and a programming model to access data. This API lets you to work with data from heterogeneous data sources, including RDBMS, entity EJBs, XML sources, web services, EIS data sources using the Java Connector Architecture, and so on. Hence you as a developer need not be familiar with a technology-specific API such as JDBC or XQuery in order to access and utilize data. Instead, you can just use SDO API.

**SDO Architecture**
In SDO, data is organized as a graph of objects, called `DataObject`. A `DataObject` is the fundamental component which is a representation of some structured data, with some properties. These properties have either a single value or multiple values, and their values can be even other data objects. Each data objects also maintains a change summary, which represents the alterations made to it.
SDO clients or consumers always use SDO programming model and API. This is generic of technology and framework, and hence the developers need not know how the underlying data they are working with is persisted. A **Data Mediator Service (DMS)** is responsible for creating a data graph from data source(s), and also for updating the data source(s) based on any changes made to a data graph. SDO clients are disconnected from both the DMS and the data source.

A DMS will create a **Data Graph**, which is a container for a tree of data objects. Another interesting fact is that a single data graph can represent data from different data sources. This is actually a design model to deal with data aggregation scenarios from multiple data sources. The data graphs form the basics of the disconnected architecture of SDO, since they can be passed across layers and tiers in an application. When doing so, they are serialized to the XML format.

A **Change Summary** contains any change information related to the data in the data object. Change summaries are initially empty and are populated as and when the data graph is modified.

**Apache Tuscany SDO**

Apache Tuscany SDO is a sub-project within open-source Apache Tuscany.

Apache Tuscany aims at defining an infrastructure that simplifies the development of Service-Oriented application networks, addressing real business problems. It is based on specifications defined by the **OASIS Open Composite Services Architecture (CSA) Member Section**, which advances open standards that simplify SOA application development.

---

For More Information:

Tuscany SDO mainly provides implementations in Java and C++. Both are available for download at: http://incubator.apache.org/tuscany/.

**SDO Sample Using Tuscany SDO**

SDO can handle heterogeneous data sources, but for the sample here, we will make use of an XML file as a data source. The sample will read as well as write an XML file, when the client program makes use of SDO API to do data operations.

**Code the Sample Artifacts**

The main artifacts for running the samples in SDO include an XSD schema file and an XML instance file. Then we have two java programs, one which reads the XML and another which creates an XML. We will look into these files first.

**hr.xsd**

The hr.xsd restricts the structure of an employee XML file, which can contain multiple employees. Each employee can have a name, address, organization, and office elements. Each of these elements can have sub-elements, which are as shown here:

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.binildas.com/apache/tuscany/sdo/sample"
targetNamespace="http://www.binildas.com/apache/tuscany/sdo/sample">
  <xsd:element name="employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="employee" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="employee">
    <xsd:annotation>
      <xsd:documentation>Employee representation</xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string" />
        <xsd:element ref="address" maxOccurs="2" />
        <xsd:element ref="organization" />
        <xsd:element ref="office" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```
<xsd:element name="organization">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:integer" />
    </xsd:complexType>
</xsd:element>

<xsd:element name="office">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="address"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:integer" />
    </xsd:complexType>
</xsd:element>

<xsd:element name="address">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="street1" type="xsd:string"/>
            <xsd:element name="street2" type="xsd:string" minOccurs="0"/>
            <xsd:element name="city" type="xsd:string"/>
            <xsd:element name="state" type="stateAbbreviation"/>
            <xsd:element ref="zip-code"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="zip-code">
    <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:pattern value="[0-9]{5}(-[0-9]{4})?"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>

<xsd:element name="stateAbbreviation">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[A-Z]{2}"/>
    </xsd:restriction>
</xsd:element>

</xsd:schema>
hr.xml

The hr.xml provided is fully constrained as per the above schema. For our sample demonstration this XML file contains data on two employees as shown here:

```xml
<?xml version="1.0"?>
<employees xmlns="http://www.binildas.com/apache/tuscany/sdo/sample">
  <employee id="30379">
    <name>Binildas C. A.</name>
    <address>
      <street1>45 Bains Compound Nanthencode</street1>
      <city>Trivandrum</city>
      <state>KL</state>
      <zip-code>695003</zip-code>
    </address>
    <organization id="08">
      <name>Software</name>
    </organization>
    <office id="31">
      <address>
        <street1>101 Camarino Ruiz</street1>
        <street2>Apt 2 Camarillo</street2>
        <city>Callifornia</city>
        <state>LA</state>
        <zip-code>93012</zip-code>
      </address>
    </office>
  </employee>
  <employee id="30380">
    <name>Rajesh R V</name>
    <address>
      <street1>1400 Salt Lake Road</street1>
      <street2>Appartment SE</street2>
      <city>Boston</city>
      <state>MA</state>
      <zip-code>20967</zip-code>
    </address>
    <organization id="15">
      <name>Research</name>
    </organization>
    <office id="21">
      <address>
        <street1>2700 Cambridge Drive</street1>
        <city>Boston</city>
        <state>MA</state>
      </address>
    </office>
  </employee>
</employees>
```
ReadEmployees.java

Now, we are going to see SDO in action. In the ReadEmployees class shown below, we first read the XML file, mentioned previously, and load it into a root DataObject. A DataObject is a graph of other DataObjects. Hence, we can iterate over the graph and get each item DataObject.

```java
public class ReadEmployees extends SampleBase{
    private static final String HR_XML_RESOURCE = "hr.xml";
    public static final String HR_XSD_RESOURCE = "hr.xsd";
    public ReadEmployees(Integer commentaryLevel) {
        super(commentaryLevel,
                SampleInfrastructure.SAMPLE_LEVEL_BASIC);
    }
    public static void main(String[] args)throws Exception{
        ReadEmployees sample = new
                ReadEmployees(COMMENTARY_FOR_NOVICE);
        sample.runSample();
    }
    public void runSample () throws Exception{
        InputStream inputStream =
                ClassLoader.getSystemResourceAsStream(HR_XML_RESOURCE);
        byte[] bytes = new byte[inputStream.available()];
        inputStream.read(bytes);
        inputStream.close();
        HelperContext scope = createScopeForTypes();
        loadTypesFromXMLSchemaFile(scope, HR_XSD_RESOURCE);
        XMLDocument xmlDoc = getXMLDocumentFromString(scope,
                new String(bytes));
        DataObject purchaseOrder = xmlDoc.getRootObject();
        List itemList = purchaseOrder.getList("employee");
        DataObject item = null;
        for (int i = 0; i < itemList.size(); i++) {
            item = (DataObject) itemList.get(i);
            System.out.println("id: " + item.get("id");
            System.out.println("name: " + item.get("name");
        }
    }
}
```
CreateEmployees.java

In the CreateEmployees class, we do the reverse process—we define DataObjects in code and build the SDO graph. At the end, the root DataObject is persisted to a file and also to the system output stream as shown in the following code.

```java
public class CreateEmployees extends SampleBase {
    private static final String HR_XML_RESOURCE_NEW = "hr_new.xml";
    public static final String HR_XSD_RESOURCE = "hr.xsd";
    public static final String HR_NAMESPACE = "http://www.binildas.com/apache/tuscany/sdo/sample";

    public CreateEmployees(Integer commentaryLevel) {
        super(commentaryLevel, SAMPLE_LEVEL_BASIC);
    }

    public static void main(String[] args) throws Exception{
        CreateEmployees sample =
            new CreateEmployees(COMMENTARY_FOR_NOVICE);
        sample.runSample();
    }

    public void runSample() throws Exception{
        HelperContext scope = createScopeForTypes();
        loadTypesFromXMLSchemaFile(scope, HR_XSD_RESOURCE);
        DataFactory factory = scope.getDataFactory();
        DataObject purchaseOrder = factory.create(HR_NAMESPACE,
            "employees");
        DataObject employee1 = purchaseOrder.createDataObject(
            "employee");
        employee1.setString("id", "3457");
        employee1.set("name", "Cindy Jones");
        DataObject homeAddress1 = employee1.createDataObject("address");
        homeAddress1.set("street1", "Cindy Jones");
        homeAddress1.set("city", "Stanchion");
        homeAddress1.set("state", "TX");
        homeAddress1.set("zip-code", "79021");
        DataObject organization1 =
            employee1.createDataObject("organization");
        organization1.setString("id", "78");
        organization1.set("name", "Sales");
        DataObject office1 = employee1.createDataObject("office");
        office1.setString("id", "43");
        DataObject officeAddress1 = office1.createDataObject("address");
        officeAddress1.set("street1", "567 Murdock");
    }
```

officeAddress1.set("street2", "Suite 543");
officeAddress1.set("city", "Millford");
officeAddress1.set("state", "TX");
officeAddress1.set("zip-code", "79025");

DataObject employee2 = purchaseOrder.createDataObject("employee");
employee2.setString("id", "30376");
employee2.set("name", "Linda Mendez");

DataObject homeAddress2 = employee1.createDataObject("address");
homeAddress2.set("street1", "423 Black Lake Road");
homeAddress2.set("street2", "Appartment 7A");
homeAddress2.set("city", "Boston");
homeAddress2.set("state", "MA");
homeAddress2.set("zip-code", "20967");

DataObject organization2 =
    employee2.createDataObject("organization");
organization2.setString("id", "78");
organization2.set("name", "HR");

DataObject office2 = employee2.createDataObject("office");
office2.setString("id", "48");

OutputStream stream = new FileOutputStream(HR_XML_RESOURCE_NEW);
scope.getXMLHelper().save(purchaseOrder, HR_NAMESPACE, "employees", stream);
stream.close();

For More Information:
www.packtpub.com/service-oriented-architecture-for-java-applications/book

outputAddress1.set("street2", "Suite 543");
outputAddress1.set("city", "Millford");
outputAddress1.set("state", "TX");
outputAddress1.set("zip-code", "79025");

DataObject employee2 = purchaseOrder.createDataObject("employee");
employee2.setString("id", "30376");
employee2.set("name", "Linda Mendez");

DataObject homeAddress2 = employee1.createDataObject("address");
homeAddress2.set("street1", "423 Black Lake Road");
homeAddress2.set("street2", "Appartment 7A");
homeAddress2.set("city", "Boston");
homeAddress2.set("state", "MA");
homeAddress2.set("zip-code", "20967");

DataObject organization2 =
    employee2.createDataObject("organization");
organization2.setString("id", "78");
organization2.set("name", "HR");

DataObject office2 = employee2.createDataObject("office");
office2.setString("id", "48");

OutputStream stream = new FileOutputStream(HR_XML_RESOURCE_NEW);
scope.getXMLHelper().save(purchaseOrder, HR_NAMESPACE, "employees", stream);
stream.close();

XMLDocument doc =
    scope.getXMLHelper().createDocument(purchaseOrder, HR_NAMESPACE, "employees");
scope.getXMLHelper().save(doc, System.out, null);
System.out.println();
}
}

Build and Run the SDO Sample
To build the sample in a single command, it is easy for you to go to ch04\sdo folder and execute the following command:

cd ch04\sdo
ant
Data and Services – All Roads Lead to Enterprise Service Bus

Now, you can execute the ReadEmployees class by executing:

```
ant read
```

![Command Prompt](image1.png)

Now, you can execute the CreateEmployees class by executing:

```
ant create
```

![Command Prompt](image2.png)

For More Information:
www.packtpub.com/service-oriented-architecture-for-java-applications/book
Service Component Architecture

We have been creating IT assets in the form of programs and codes since many years, and been implementing SOA architecture. This doesn't mean that we follow a big bang approach and throw away all old assets in place of new. Instead, the success of any SOA effort depends largely on how we can make the existing assets co-exist with new architecture principles and patterns. To this end, Service Component Architecture (SCA) aims at creating new and transforms existing, IT assets into re-usable services more easily. These IT assets can then be rapidly adapted to changing business requirements. In this section, we will introduce SCA and also look into some working samples for the same.

What is SCA?

SCA introduces the notion of services and references. A component which implements some business logic offers their capabilities through service-oriented interfaces. Components may also consume functionality offered by other components through service-oriented interfaces, called service references. If you follow SOA best practices, you will perhaps appreciate the importance of fine-grained tight coupling and coarse-grained loose coupling between components. SCA composition aids recursive assembly of coarse-grained components out of fine-grained tightly coupled components. These coarse-grained components can even be recursively assembled to form higher levels of coarse-grained components. In SCA, a composite is a recursive assembly of fine-grained components. All these are shown in the SCA assembly model in the following screenshot:

[Diagram of SCA assembly model]

For More Information:
www.packtpub.com/service-oriented-architecture-for-java-applications/book
Apache Tuscany SCA Java
Apache Tuscany SCA is a sub-project within open-source Apache Tuscany, which has got a Java implementation of SCA. Tuscany SCA is integrated with Tomcat, Jetty, and Geronimo.

SCA Java runtime is composed of core and extensions. The core wires functional units together and provides SPIs that extensions can interact with. Extensions enhance SCA runtime functionality such as service discovery, reliability, support for transport protocols, and so on.


SCA Sample Using Tuscany SCA Java
The sample here provides a single booking service with a default SCA (java) binding. The BookingAgentServiceComponent exercises this component by calling three other components that is, FlightServiceComponent, HotelServiceComponent, and CabServiceComponent as shown in the BookingAgent SCA assembly diagram shown below:

Code the Sample Artifacts
The sample consists of two sets of artifacts. The first set is the individual fine-grained service components. The second set is the coarse-grained service component, which wires the referenced fine-grained service components.
Chapter 4

Code Fine-Grained Service Components
There are three fine-grained service components whose code is self explanatory and are listed below:

FlightServiceComponent

```java
public interface IFlightService{
    String bookFlight(String date, int seats, String flightClass);
}

class FlightServiceImpl implements IFlightService{
    public String bookFlight(String date, int seats, String flightClass){
        System.out.println("FlightServiceImpl.bookFlight...");
        return "Success";
    }
}
```

HotelServiceComponent

```java
public interface IHotelService{
    String bookHotel(String date, int beds, String hotelClass);
}

class HotelServiceImpl implements IHotelService{
    public String bookHotel(String date, int beds, String hotelClass){
        System.out.println("HotelServiceImpl.bookHotel...");
        return "Success";
    }
}
```

CabServiceComponent

```java
public interface ICabService{
    String bookCab(String date, String cabType);
}

class CabServiceImpl implements ICabService{
    public String bookCab(String date, String cabType){
        System.out.println("CabServiceImpl.bookCab...");
        return "Success";
    }
}
```
Data and Services – All Roads Lead to Enterprise Service Bus

**Code BookingAgent Service Component**

BookingAgentServiceComponent depends on three referenced service components, which are the fine-grained service components listed previously. They are initialized by the dependency injection by the SCA runtime. Also, for the actual business method invocation, the call is delegated to the referenced service components as shown in the bookTourPackage method in the following code:

```java
import org.osoa.sca.annotations.Reference;
public class BookingAgentServiceComponent implements IBookingAgent{
    private IFlightService flightService;
    private IHotelService hotelService;
    private ICabService cabService;
    @Reference
    public void setFlightService(IFlightService flightService) {
        this.flightService = flightService;
    }
    @Reference
    public void setHotelService(IHotelService hotelService) {
        this.hotelService = hotelService;
    }
    @Reference
    public void setCabService(ICabService cabService) {
        this.cabService = cabService;
    }
    public String bookTourPackage(String date,
       int people, String tourPack){
        System.out.println("BookingAgent.bookTourPackage...");
        String flightBooked = 
        flightService.bookFlight(date, people, tourPack);
        String hotelBooked = 
        hotelService.bookHotel(date, people, tourPack);
        String cabBooked = 
        cabService.bookCab(date, tourPack);
        if((flightBooked.equals("Success")) &&
           (hotelBooked.equals("Success")) &&
           (cabBooked.equals("Success"))){
            return "Success";
        } else{
            return "Failure";
        }
    }
}
```
Code BookingAgent Client

The BookingAgentClient first creates an instance of SCADomain and then gets a reference of the BookingAgentServiceComponent using the name of the configured service component. Then it executes the business method, bookTourPackage.

```java
import org.apache.tuscany.sca.host.embedded.SCADomain;
pubic class BookingAgentClient{
    public static void main(String[] args) throws Exception {
        SCADomain scaDomain =
            SCADomain.newInstance("BookingAgent.composite");
        IBookingAgent bookingAgent =
            scaDomain.getService(IBookingAgent.class,
                "BookingAgentServiceComponent");
        System.out.println("BookingAgentClient.bookingTourPackage...");
        String result = bookingAgent.bookTourPackage(
            "20Dec2008", 5, "Economy");
        System.out.println("BookingAgentClient.bookedTourPackage : ",
            result);
        scaDomain.close();
    }
}
```

Build and Run the SCA Sample

To build the sample in a single command, it is easy for you to go to ch04\sca folder and execute the following command:

```bash
cd ch04\sca
ant
```

Now, you can execute the BookingAgentClient program by executing:

```bash
ant run
```
You can see that the BookingAgentServiceComponent will delegates calls to book individual line items to the referred service components and if all the individual bookings are done right, the overall transaction is "success". The following figure shows the screenshot of such a success scenario:

![Command Prompt screenshot]

**Message-Oriented Middleware**

Having seen some of the newer technology approaches in integrating data and services, we now need to move on to the next stage of discussion on the different platform level services available for integration. *message-oriented middleware* (MOM) is the main aspect we need to discuss in this context.

**What is MOM?**

Just like using sockets for *Inter-Process Communications* (IPC), we use messaging when multiple processes need to communicate with each other to share data. Of course we can get the same effect when we use files or use a shared database for data level integration. But at times we may also require other *Quality of service* (QoS) features, a few amongst them will be described later. Thus, a MOM manages the movement of messages between systems connected by a network in a reliable fashion by separating the message sending step from the message receiving step so that the message exchange takes place in a loosely coupled and detached manner. The dynamics of message delivery in a MOM is shown in the following figure:

---

For More Information:  
www.packtpub.com/service-oriented-architecture-for-java-applications/book
Here, the message delivery happens in the following steps:

- The sender process will just 'fire and forget' the message.
- The MOM will 'store and forward' the message.
- The receiver process will 'asynchronously receive' the message later.
Since the entire process happens in stages, even if one of the players in one of these stages is not ready for the message transmission, it won't affect the previous stages or the players involved there.

**Benefits of Using MOM**

MOM will have a set of features which makes it different from other style of communications such as RPC or Sockets, which may be required by some class of applications. Let us now look into some of these features.

- **Asynchronous Style of communication**: In MOM communications, a sender application after sending the message need not wait for either the sending of the message to complete or to get a response from the receiving applications. Both these after effects can be affected later, perhaps in a different thread of execution. This will increase application responsiveness.

- **Platform or Language level interoperability**: The world is never ideal, and we never have the luxury to always work with cutting edge technologies alone or to chose the platform or language of choice of all interconnecting systems. Sometimes there may be legacy systems, while sometimes there may be SOA-based web service interfaces to interconnect. Whatever be the case, a MOM allows them all to communicate through a common messaging paradigm. This universal connectivity is sometimes called as the Message Bus pattern.

- **Application down times**: Applications interconnecting together can sit in any geography or in any time zone, and all of them will have their own down times too. Hence, if a sender application sends some message to a receiver and the receiving application is not up at that time, the message shouldn't get lost. Further, when the receiver comes up the next time, it should receive the message once and exactly once. An MOM, with it's store and forward capability will give the maximum flexibility for interacting systems to exchange messages at their own pace.

- **Peak time processing and Throttling**: For a receiving application, there may be peak hours of the day during which it cannot process further request messages. Any further processing might degrade even the undergoing request processing. Hence, some kind of admission control or queuing up of additional requests to be processed further is required. Such mechanisms are the norm for a MOM with its store queues.
- **Reliability**: Message stores are introduced at multiple stages in the message delivery path. At the sender's end and at the receiver's end, message stores will guarantee staged message delivery which guarantees message reliability in stages. So, if a step in the message delivery fails, the step can be replayed retrieving the message again from the previous step (or previous stage message store).

- **Mediating services**: By using a MOM, an application becomes disconnected from the other applications. One application needs to reconnect only to the messaging system, not to all the other messaging applications it need to interconnect with. The applications are thus loosely coupled, still interconnected.

All the above features distinguish MOM from its counterpart styles of message interactions, which we leverage in many architectural patterns such as the Enterprise Service Bus, which we shall describe next.

### Enterprise Service Bus

**Enterprise Service Bus (ESB)** is an architectural style for integrating systems, applications, and services. ESB provides a technology stack which acts like an integration bus to which multiple applications can talk. So, if two or more applications need to talk to each other, they don't need to integrate directly, but only need to talk to the ESB. The ESB will do the mediation services on behalf of the communicating applications, which may or may not be transparent to these communicating applications.

### EAI and ESB

In order to understand ESB better, we need to understand the technical context under which we have to discuss this concept. The context is **Enterprise Application Integration (EAI)**, which deals with the sharing of data and processes, amongst connected systems in an enterprise. Traditionally, we have been doing EAI to do integration. EAI defines connection points between systems and applications. But when we consider integration in the context of SOA, we need to think more than just integration—we need to think in terms of services and service-based integration. Services expose standards, and if there is a way to leverage this standardization in services in defining the integration points too, then it would open up new possibilities in terms of standard connectors and adaptors.
Before we get into the details of ESB, it makes sense to compare and contrast it with other integration architectures as well. In EAI, the Point-to-Point, and the Hub and Spoke architectures are frequently used in many bespoke solutions. They are used in many vendor products too. These architectures are schematically shown in the following figure:

In Point-to-point, we define an integration solution for a pair of applications. At the integration points, we have tight coupling, since both ends have knowledge about their peers. Each peer combination need to have its own set of connectors. Hence, the number of connectors increases as the number of applications increases. Whereas in the Hub and Spoke architecture, we have a centralized hub (or broker) to which all applications are connected. Each application connects with the central hub through lightweight connectors. The lightweight connectors facilitates for application integration with minimum or zero changes to the existing applications.
Now, we will look into the Enterprise Message Bus and the Enterprise Service Bus architectures.

The Enterprise Message Bus makes use of the MOM stack and toolset to provide a common messaging backbone for applications to interconnect. Sometimes, the applications have to use adapter which handles scenarios such as invoking CICS transactions. Such an adapter may provide connectivity between the applications and the message bus using proprietary bus APIs, and application APIs.

When you move from a traditional MOM to the ESB-based architecture, the major difference is that the applications communicate through a Service Oriented Architecture (SOA) backbone. This backbone is again built over the common MOM, but it provides Intelligent Connectors. These Intelligent Connectors are abstract in the sense that they only define the transport binding protocols and service interface, not the real implementation details. They are intelligent, because they have logic built-in along with the ESB to selectively bind to services at run time. This capability enhances agility for applications by allowing late binding of services and deferred service choice. Moreover, since these intelligent connectors are deployable in the ESB runtime, they are even available as COTS (component off the shelf) libraries. This means that the ESB will open up a market for vendors to build and sell connectors for proprietary EIS systems, which will expose standard interfaces outside the ESB.
Java Business Integration

Java Business Integration (JBI) provides a collaboration framework which provides standard interfaces for integration components and protocols to plug into, thus allowing the assembly of Service Oriented Integration (SOI) frameworks following the ESB pattern. JBI is based on JSR 208, which is an extension of Java 2 Enterprise Edition (J2EE) and is specific for JBI Service Provider Interfaces (SPI). SOA and SOI are the targets of JBI and hence it is built around WSDL. Integration components can be plugged into the JBI environment using a service model-based on WSDL.

For readers who would like to delve deep into Java Business Integration, you are advised to refer to "Service Oriented Java Business Integration" by Binildas A. Christudas ISBN: 1847194400 published by Packt Publishing, since we cannot cover such a vast topic in a single section or in a single chapter in a book.

OpenESB

Project OpenESB is an open-source implementation of JSR-208 (JBI) hosted by Java.net community and is available for download at https://open-esb.dev.java.net/. OpenESB allows easy integration of web services thus creating loosely coupled enterprise class composite applications.

OpenESB Architecture provides the following salient features, which distinguishes itself from other closed ESB solutions available in the market today:

- **Application Server support**: OpenESB has got good integration with Glassfish application server, thus enabling the integration components to leverage the reliability, scalability, resiliency, deployment, and management capabilities of the application server.

- **Composite application support**: In OpenESB, we can use BPEL and similar composite application support tools to create composite applications which are self-contained artifacts that contain other sub-artifacts.

- **Composite Application Editor**: OpenESB comes with Composite Application Editor that helps the user 'wire-together' and create new Composite Applications from fine-grained services.

- **JBI Bus**: The JBI Bus provides a pluggable infrastructure which can host a set of integration component, which can integrate various types of IT assets in the enterprise. The JBI bus provides an in-memory messaging bus called the Normalized Message Router (NMR). It is through this NMR the messages which are normalized and in standard abstract WSDL format flows.
• **Service Engines and Binding Components**: JBI supports two types of components, such as Service Engines and Binding Components. Service Engines provide business logic and transformation services to other components, as well as consume such services. Binding components provide services external to the OpenESB environment available to the NMR.

• **Business Logic Units**: These are processing units similar to a BPEL component which can orchestrate the services available at the ESB and provide higher level of business process functionality again at the ESB.

• **Global Service Collaboration Networks**: OpenESB supports for a services fabric style of service assembly which is a kind of service virtualization which divides the organization's information assets into "Virtual Services" without regard to the transport or binding protocol of the actual implementation components.

• **Monitoring**: OpenESB also provides the ability to centrally monitor, manage, and administer the distributed runtime system of the ESB.

Another noticeable factor which advocates the popularity of OpenESB is the huge list of components and library support available in the industry which can plug easily into the OpenESB JBI infrastructure, a part of which is shown in the following screenshot taken from the OpenESB website:
Summary

SOA is not a single product or single reference architecture to be followed, but is all about best practices, reference architectures, processes, toolsets, and frameworks along with many other things which will help you and your organization increase the responsiveness and agility of your enterprise architecture. Standards and frameworks play a greater role in enabling easy and widespread industry adoption of SOA. In this chapter, you have seen few emerging standards such as SDO and SCA, addressing from data integration till service and component integration. Newer architectural patterns such as ESB and Data Services provide you with a wider framework upon which you can enable your integration points for open and flexible information flow. In the next chapter, we will specifically look more into integration with emphasis on these new architectural styles and patterns.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.