CHAPTER 4

XML Processing

WEB service applications often pass information using XML documents. Application developers whose applications accept XML documents must understand how best to extract information from these XML documents and use that information in their business processing. They must also understand how to assemble XML documents from the results of this business processing.

This chapter provides an extensive coverage of XML document handling. To make it more manageable, you may want to concentrate on the sections of particular interest to you. These sections are as follows:

- "XML Overview" on page 122—Provides an overview of basic XML concepts. If you are not familiar with XML, this section provides a concise summary of key XML concepts and technologies. Even if you do know XML well, you may want to skim this part since it highlights issues important for designing XML-based applications.
- "Outline for Handling XML Documents" on page 128—Describes how applications typically handle XML documents.
- "Designing XML-Based Applications" on page 131—Covers such design topics as receiving and sending XML documents, validating XML documents, mapping XML documents to data objects, applying business logic on documents or objects, and keeping business processing logic separate from XML processing logic. This discussion also includes recommendations that help determine how to best design an XML-based application. Architects and application designers should find this section of particular interest.

- "Implementing XML-Based Applications" on page 164—Provides guidelines for developers on how best to implement XML-based applications. This section includes an in-depth discussion of the various XML-processing technologies that developers can use to implement their applications. It covers the advantages and disadvantages of the principal XML programming models and provides recommendations for when you should consider using each of the models.
- "Performance Considerations" on page 182—Outlines guidelines for maximizing performance. This section makes specific recommendations for approaches developers can take to keep performance at an acceptable level.

Many of these concepts for designing XML-based applications are generic in nature; they apply to any application handling XML documents. Since Web services are XML-based applications, these concepts especially apply to the design of Web service endpoints and clients. The chapter emphasizes the design considerations that should be kept in mind when developing Web service endpoints and clients.

Although it presents the basic XML concepts, this chapter assumes that you have a working knowledge of the XML processing technologies, especially SAX, DOM, XSLT, and JAXB. (Refer to Chapter 2 for more details on these technologies.) Whenever possible, the chapter uses the scenarios introduced in Chapter 1 to illustrate various points.

4.1 XML Overview

While you may already be familiar with XML, it is important to understand XML concepts from the point of view of applications handling XML documents. With this knowledge, you are in a better position to judge the impact of your design decisions on the implementation and performance of your XML-based applications.

Essentially, XML is a markup language that enables hierarchical data content extrapolated from programming language data structures to be represented as a marked-up text document. As a markup language, XML uses tags to mark pieces of data. Each tag attempts to assign meaning to the data associated with it; that is, transform the data into information. If you know SGML (Standard Generalized Markup Language) and HTML (HyperText Markup Language), then XML will look familiar to you. XML is derived from SGML and also bears some resemblance to HTML, which is also a subset of SGML. But unlike HTML, XML

focuses on representing data rather than end-user presentation. While XML aims to separate data from presentation, the end-user presentation of XML data is nevertheless specifically addressed by additional XML-based technologies in rich and various ways.

Although XML documents are not primarily intended to be read by users, the XML specification clearly states as one of its goals that "XML documents should be human-legible and reasonably clear." This legibility characteristic contributed to XML's adoption. XML supports both computer and human communications, and it ensures openness, transparency, and platform-independence compared to a binary format.

A grammar along with its vocabulary (also called a schema in its generic acception) defines the set of tags and their nesting (the tag structure) that may be allowed or that are expected in an XML document. In addition, a schema can be specific to a particular domain, and domain-specific schemas are sometimes referred to as markup vocabularies. The Document Type Definition (DTD) syntax, which is part of the core XML specification, allows for the definition of domain-specific schemas and gives XML its "eXtensible" capability. Over time, there have been an increasing number of these XML vocabularies or XML-based languages, and this extensibility is a key factor in XML's success. In particular, XML and its vocabularies are becoming the lingua franca of business-to-business (B2B) communication.

In sum, XML is a metalanguage used to define other markup languages. While tags help to describe XML documents, they are not sufficient, even when carefully chosen, to make a document completely self-describing. Schemas written as DTDs, or in some other schema language such as the W3C XML Schema Definition (XSD), improve the descriptiveness of XML documents since they may define a document's syntax or exact structure. But even with the type systems introduced by modern schema languages, it is usually necessary to accompany an XML schema with specification documents that describe the domain-specific semantics of the various XML tags. These specifications are intended for application developers and others who create and process the XML documents. Schemas are necessary for specifying and validating the structure and, to some extent, the content of XML documents. Even so, developers must ultimately build the XML schema's tag semantics into the applications that produce and consume the documents. However, thanks to the well-defined XML markup scheme, intermediary applications such as document routers can still handle documents partially or in a generic way without knowing the complete domain-specific semantics of the documents.

The handling of the following XML document concepts may have a significant impact on the design and performance of an XML-based application:

- **Well-formedness**—An XML document needs to be well formed to be parsed. A well-formed XML document conforms to XML syntax rules and constraints, such as:
  - The document must contain exactly one root element, and all other elements are children of this root element.
  - All markup tags must be balanced; that is, each element must have a start and an end tag.
  - Elements may be nested but they must not overlap.
  - All attribute values must be in quotes.
- **Validity**—According to the XML specification, an XML document is considered valid if it has an associated DTD declaration and it complies with the constraints expressed in the DTD. To be valid, an XML document must meet the following criteria:
  - Be well-formed
  - Refer to an accessible DTD-based schema using a Document Type Declaration: `<!DOCTYPE>`
  - Conform to the referenced DTD

  With the emergence of new schema languages, the notion of validity is extended beyond the initial specification to other, non-DTD-based schema languages, such as XSD. For these non-DTD schemas, the XML document may not refer explicitly to the schema, though it may only contain a hint to the schema to which it conforms. The application is responsible for enabling the validation of the document. Regardless of any hints, an application may still forcefully validate this document against a particular schema. (See “Validating XML Documents” on page 139.)
- **Logical and physical forms**—An XML document has one logical form that may be laid out potentially in numerous physical forms. The physical form (or forms) represent the document’s storage layout. The physical form consists of storage units called entities, which contain either parsed or unparsed data. Parsed entities are invoked by name using entity references. When parsed, the reference is replaced by the contents of the entity, and this replacement text be-

comes an integral part of the document. The logical form is the entire document regardless of its physical or storage layout.

An XML processor, in the course of processing a document, may need to find the content of an external entity—this process is called entity resolution. The XML processor may know some identifying information about the external entity, such as its name, system, or public identifier (in the form of a URI: URL or URN), and so forth, which it can use to determine the actual location of the entity. When performing entity resolution, the XML processor maps the known identifying information to the actual location of the entity. This mapping information may be accessible through an entity resolution catalog.

4.1.1 Document Type and W3C XML Schema Definitions

Originally, the Document Type Definition (DTD) syntax, which is part of the core XML 1.0 specification and became a recommendation in 1998, allowed for the definition of domain-specific schemas. However, with the growth in the adoption of XML (particularly in the B2B area), it became clear that the DTD syntax had some limitations. DTD's limitations are:

- It uses a syntax that does not conform to other XML documents.
- It does not support namespaces. However, with some cleverness, it is possible to create namespace-aware DTD schemas.
- It cannot express data types. With DTD, attribute values and character data in elements are considered to be text (or character strings).

To address these shortcomings, the W3C defined the XML Schema Definition language (XSD). (XSD became an official recommendation of the W3C in 2001.) XSD addresses some of the shortcomings of DTD, as do other schema languages, such as RELAX-NG. In particular, XSD:

- Is itself an application of XML based on the XML specification—An XML schema can be written and manipulated just like an XML document.
- Supports namespaces—By supporting namespaces, XSD allows for modular schema design and permits the composition of XSD schema definitions. It particularly solves the problem of conflicting tag names, which can often occur with modularization.
- Supports data types—XSD provides a type system that supports type deriva-

tion and restriction, in addition to supporting various built-in simple types, such as integer, float, date, and time.

The following convention applies to the rest of the chapter: The noun "schema" or "XML schema" designates the grammar or schema to which an XML document must conform and is used regardless of the actual schema language (DTD, XSD, and so forth). **Note:** While XSD plays a major role in Web services, Web services may still have to deal with DTD-based schemas because of legacy reasons.

As an additional convention, we use the word "serialization" to refer to XML serialization and deserialization. We explicitly refer to Java serialization when referring to serialization supported by the Java programming language. Also note that we may use the terms "marshalling" and "unmarshalling" as synonyms for XML serialization and deserialization. This is the same terminology used by XML data-binding technologies such as JAXB.

4.1.2 XML Horizontal and Vertical Schemas

XML schemas, which are applications of the XML language, may apply XML to horizontal or vertical domains. Horizontal domains are cross-industry domains, while vertical domains are specific to types of industries. Specific XML schemas have been developed for these different types of domains, and these horizontal and vertical applications of XML usually define publicly available schemas.

Many schemas have been established for horizontal domains; that is, they address issues that are common across many industries. For example, W3C specifications define such horizontal domain XML schemas or applications as Extensible HyperText Markup Language (XHTML), Scalable Vector Graphics (SVG), Mathematical Markup Language (MathML), Synchronized Multimedia Integration Language (SMIL), Resource Description Framework (RDF), and so forth.

Likewise, there are numerous vertical domain XML schemas. These schemas or applications of XML define standards that extend or apply XML to a vertical domain, such as e-commerce. Typically, groups of companies in an industry develop these standards. Some examples of e-commerce XML standards are Electronic Business with XML (ebXML), Commerce XML (CXML), Common Business Language (CBL), and Universal Business Language (UBL).

When designing an enterprise application, developers often may define their own custom schemas. These custom schemas may be kept private within the enterprise. Or, they may be shared just with those partners that intend to exchange

data with the application. It is also possible that these custom schemas may be publicly exposed. Such custom schemas or application-specific schemas are defined either from scratch or, if appropriate, they may reuse where possible existing horizontal or vertical schema components. Note that publishing schemas in order to share them among partners can be implemented in various ways, including publishing the schemas along with Web service descriptions on a registry (see "Publishing a Web Service" on page 101).

4.1.3 Other Specifications Related to XML

For those interested in exploring further, here is a partial list of the many specifications that relate to XML.

- Document Object Model (DOM)—The Document Object Model is an interface, both platform and language neutral, that lets programs and scripts dynamically process XML documents. Using DOM, programs can access and update the content, structure, and style of documents.
- Xpath—The Xpath specification defines an expression language for navigating and processing an XML source document, including how to locate elements in an XML document.
- eXtensible Stylesheet Language Transformations (XSLT)—This specification, which is a subset of eXtensible Stylesheet Language (XSL), describes how to transform XML documents between different XML formats as well as non-XML formats.
- Namespaces—This specification describes how to associate a URI with tags, elements, attribute names, and data types in an XML document, to resolve ambiguity when elements and attributes have the same names.
- XML Information Set—This specification, often referred to as Infoset, provides the definitions for information in XML documents that are considered well formed according to the Namespaces criteria.
- Canonical XML—This specification addresses how to resolve syntactic variations between the XML 1.0 and the Namespaces specifications to create the physical, canonical representation of an XML document.

4.2 Outline for Handling XML Documents

An XML-based application, and in particular a Web service application, may consume or produce XML documents, and such an application may implement three distinct processing phases:

1. The application consumes an XML document.
2. The application applies its business logic on information retrieved from the document.
3. The application produces an XML document for response.

Generally speaking, an XML-based application exposes an interface that is defined in terms of the schemas for the XML documents that it consumes and produces as well as by the communication or interaction policies of the application. In the case of a Web service, a Web Services Description Language (WSDL) document describes this interface, and this document refers to the XML schemas to which the exchanged documents conform.

Let's examine the steps for handling XML documents. (See Figure 4.1.) The first phase, consuming XML incoming documents or XML input processing, consists of the following steps:

1. **Parse and optionally validate the document**—Parse and validate the incoming document against the schema to which the document should conform.
2. **Transform the document**—If necessary, transform the document from its conforming schema to the internally supported schema.
3. **Disassemble the document or pass it as is to the next phase**—Disassembling a document consists of these steps:
   - Search the incoming document for relevant information. The information can be recognized by using the embedded tags or its expected location within the document.
   - Extract the relevant information once it is located.
   - Optionally, map the retrieved information to domain-specific objects.

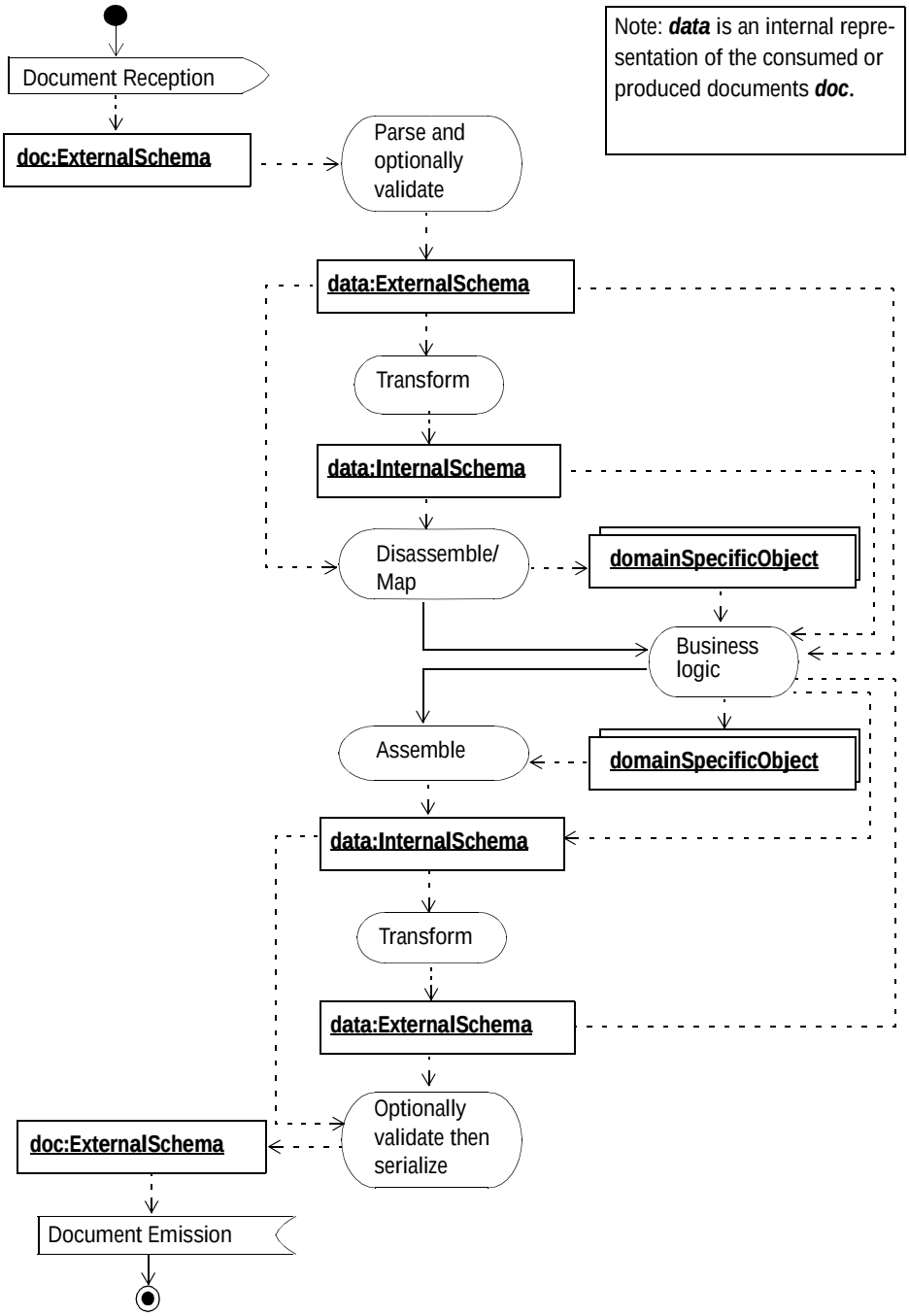

Figure 4.1 Activity Diagram Modeling Typical Steps for Handling XML Documents

In the second phase, the application applies the business logic. This entails actually processing the input information retrieved from the XML document. Such processing is considered document-centric when the logic handles the document directly, or object-centric when the logic is applied to domain-specific objects to which the retrieved information has been mapped. As a result of this processing, the application may generate XML output information, which is the third phase.

The steps for this third phase, XML output processing, mirror that of XML input processing. Producing XML output consists of the following steps:

1. **Assemble the document**—Assemble the output document from the results of applying the application's logic.
2. **Transform the document**—If necessary, transform the document from the internally supported schema to the appropriate external vertical schemas.
3. **Optionally validate, then serialize, the document**—Validating prior to serializing the output document is often considered optional.

It is quite possible that an XML-based application may only implement some of these phases, generally those that apply to its services. For example, some applications use XML just for configuration purposes, and these applications may only consume XML documents. Other applications just generate device-targeted content, and these applications may only produce XML documents (Wireless Markup Language, HTML, SVG, and so forth). E-commerce applications, on the other hand, may both consume and produce XML documents.

There are also applications that specialize in particular operations, such as those that specialize in transformations, in which case they perform only their intended operations. For instance, an application's function might be to select and apply a transformation to a document based on the processing requirements passed with the document, thus producing one or more new documents. In this example, the application's logic may consist only of the style sheet selection logic.

Developers implementing applications with a document-centric processing model—where the business logic itself handles the documents—may find that document handling is more entangled with the business logic. In particular, the steps that may intermingle with the business logic are those to disassemble the consumed documents and assemble the output document. These cases require careful handling. See "Abstracting XML Processing from Application Logic" on page 155.

Clients, whether a Web service peer or a rich client interacting with an end user, implement processes, such as the process just presented, for handling XML documents either submitted along with requests to a service or received as responses. Sometimes the application logic of human-facing rich clients may have to deal with the presentation of received documents to the end user rather than with business logic processing.

Note that real-world enterprise applications may have several instances of this abstract process to handle documents with unrelated schemas or documents with unrelated purposes. Moreover, these processes can actually be implemented to span one or more layers of an enterprise application. In particular, in the case of a Web service endpoint, these phases can be implemented in both the interaction and processing layers, depending on the processing model used by the Web service. See the discussion on a Web services layered architecture in "Key Web Services Design Decisions" on page 61.

4.3 Designing XML-Based Applications

There are a number of considerations to keep in mind when designing XML-based applications, particularly Web service applications. For one, you may need to design an XML schema specific for your domain. You also need to consider how your application intends to receive and send documents, and how and when to go about validating those documents. It is also important to separate the XML document processing from the application's business logic processing. ("Choosing Processing Models" on page 151 discusses in more detail separating XML document from business logic processing.)

Whether you design your own domain-specific schema or rely on standard vertical schemas, you still must understand the dynamics of mapping the application's data model to the schema. You also need to consider the processing model, and whether to use a document-centric model or an object-centric model.

These issues are discussed in the next sections.

4.3.1 Designing Domain-Specific XML Schemas

Despite the availability of more and more vertical domain schemas, application developers still may have to define application-specific XML schemas that must be agreed upon and shared between interoperating participants. With the introduction of modern schema languages such as XSD, which introduced strong data typing and

type derivation, XML schema design shares many of the aspects of object-oriented design especially with respect to modularization and reuse.

The design of domain-specific XML schemas breaks down according to the definition of XML schema types, their relationship to other types, and any constraints to which they are subjected. The definitions of such XML schema types, relationships, and constraints are typically the result of the analysis of the application domain vocabulary (also called the business vocabulary). As much as possible, schema designers should leverage already-defined public vertical domain schema definitions to promote greater acceptance and interoperability among intended participants. The designers of new schemas should keep interoperability concerns in mind and try to account for reuse and extensibility. Figure 4.2 shows the UML model of a typical XML schema.

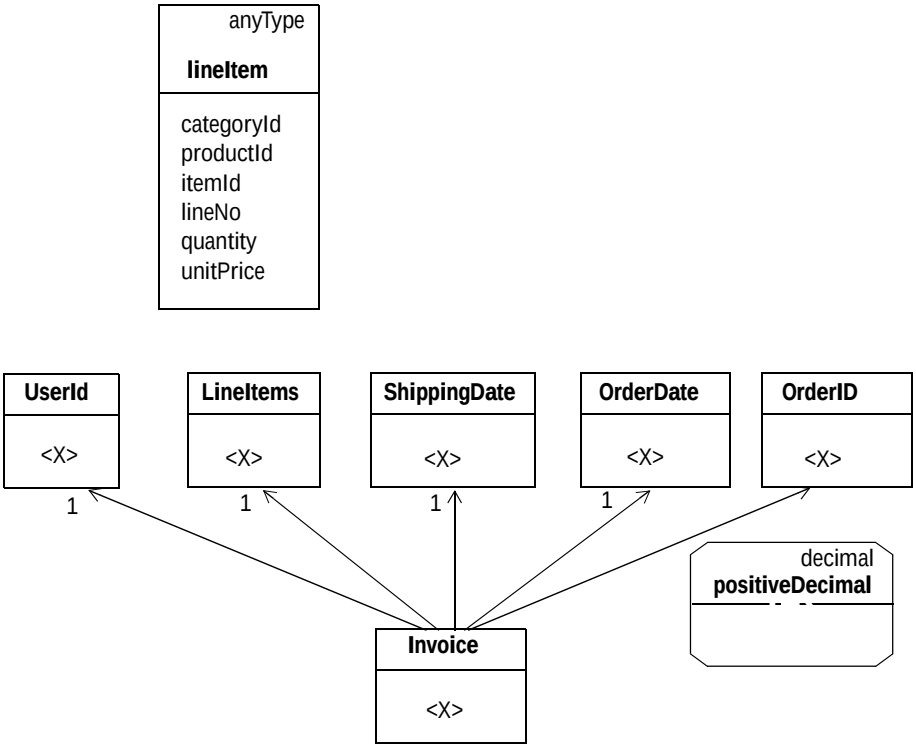

Figure 4.2 Model for an XML Schema (`Invoice.xsd`)

The strong similarity between object-oriented design and XML schema design makes it possible to apply UML modelling when designing XML schemas. Designers can use available software modelling tools to model XML schemas as UML class diagrams and, from these diagrams, to generate the actual schemas in a target schema language such as XSD. Code Example 4.1 shows an example of a schema based on XSD.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" ...>
    <xsd:element name="Invoice">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="OrderId" type="xsd:string" />
                ...
                <xsd:element name="ShippingDate" type="xsd:date" />
                <xsd:element name="LineItems">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element type="lineItem" minOccurs="1"
                            maxOccurs="unbounded" />
                        </xsd:sequence>
                    </xsd:complexType>
                    <xsd:unique name="itemIdUniqueness">
                        <xsd:selector xpath="LineItem"/>
                        <xsd:field xpath="@itemId"/>
                    </xsd:unique>

                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="lineItem">
        <xsd:attribute name="categoryId" type="xsd:string"
        use="required" />
        ...
        <xsd:attribute name="unitPrice" type="positiveDecimal"
        use="required" />
```

```
    </xsd:complexType>

    <xsd:simpleType name="positiveDecimal">
        <xsd:restriction base="xsd:decimal">
            <xsd:minInclusive value="0.0" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

Code Example 4.1 An Invoice XSD-Based Schema (`Invoice.xsd`)

To illustrate, consider the Universal Business Language (UBL) library, which provides a standard library of XML business documents, such as purchase orders, invoices, and so forth. UBL is a conceptual model of a collection of object classes and associations, called business information entities (BIES). These entities are organized into specific hierarchies, from which specific document types are assembled. As a result, UBL is:

- An XML-based business language
- Built on existing electronic data interchange (EDI) and XML business-to-business schemas or vocabularies
- Applicable across industry sectors and electronic trade domains
- Designed to be modular, reusable, and extensible

Additionally, as with any software design, there must be a balance between reusability, maintainability, and performance. This holds true both for the design of the XML schema itself and the logical and physical layout of the XML documents or schema instances. For example, consider a schema that reuses type and element definitions from other schemas. Initially loading this schema may require numerous network connections to resolve these external definitions, resulting in a significant performance overhead. Although this issue is well understood, and some XML-processing technologies may provide solutions in the form of XML entity catalogs, the developer may have to explicitly address this issue. Similarly, dynamically-generated instance of a document may be laid out such that it uses external entity references to include static or less dynamic fragments rather than embedding these fragments. This arrangement may potentially require the con-

sumer of this document to issue many network connections to retrieve these different fragments. Although this sort of modularization and inclusion may lead to significant network overhead, it does allow consumers of document schemas and instances to more finely tune caching mechanisms. See "Performance Considerations" on page 182.

Generally, document schema design and the layout of document instances closely parallel object-oriented design. In addition, design strategies exist that identify and provide well-defined solutions to common recurring problems in document schema design.

Keep the following recommendations in mind when designing an XML schema:

- ❒ Adopt and develop design patterns, naming conventions, and other best practices similar to those used in object-oriented modelling to address the issues of reuse, modularization, and extensibility.
- ❒ Leverage existing horizontal schemas, and vertical schemas defined within your industry, as well as the custom schemas already defined within your enterprise.
- ❒ Do not solely rely on self-describing element and attribute names. Comment and document custom schemas.
- ❒ Use modelling tools that support well-known schema languages such as XSD.

Keep in mind that reusing schemas may enable the reuse of the corresponding XML processing code.

4.3.2 Receiving and Sending XML Documents

XML schemas of documents to be consumed and produced are part of the overall exposed interface of an XML-based application. The exposed interface encompasses schemas of all documents passed along with incoming and outgoing messages regardless of the message-passing protocol—SOAP, plain HTTP, or JMS.

Typically, an application may receive or return XML documents as follows:

- Received through a Web service endpoint: either a JAX-RPC service endpoint or EJB service endpoint if the application is exposed as a Web service. (See Chapter 3 for more information.)

- Returned to a Web service client: if the application is accessing a Web service through JAX-RPC. (See Chapter 5 for more details.)
- Through a JMS queue or topic (possibly attached to a message-driven bean in the EJB tier) when implementing a business process workflow or implementing an asynchronous Web service architecture. (See "Delegating Web Service Requests to Processing Layer" on page 92.)

Note that a generic XML-based application can additionally receive and return XML documents through a servlet over plain HTTP.

Recall from Chapter 3 that a Web service application must explicitly handle certain XML schemas—schemas for SOAP parameters that are not bound to Java objects and schemas of XML documents passed as attachments to SOAP messages. Since the JAX-RPC runtime passes SOAP parameter values (those that are *not* bound to Java objects) as `SOAPElement` document fragments, an application can consume and process them as DOM trees—and even programmatically bind them to Java objects—using XML data-binding techniques such as JAXB. Documents might be passed as attachments to SOAP messages when they are very large, legally binding, or the application processing requires the complete document infoset. Documents sent as attachments may also conform to schemas defined in languages not directly supported by the Web service endpoint.

Code Example 4.2 and Code Example 4.3 illustrate sending and receiving XML documents.

```
public class SupplierOrderSender {
    private SupplierService_Stub supplierService;

    public SupplierOrderSender(URL serviceEndPointURL) {
        // Create a supplier Web service client stub
        supplierService = ...
        return;
    }
    // Submits a purchase order document to the supplier Web service
    public String submitOrder(Source supplierOrder)
        throws RemoteException, InvalidOrderException {
        String trackingNumber
            = supplierService.submitOrder(supplierOrder);
        return trackingNumber;
```

```
    }
}
```

Code Example 4.2 Sending an XML Document Through a Web Service Client Stub

```
public class SupplierServiceImpl implements SupplierService, ... {

    public String submitOrder(Source supplierOrder)
        throws InvalidOrderException, RemoteException {
        SupplierOrderRcvr supplierOrderRcvr
            = new SupplierOrderRcvr();
        // Delegate the processing of the incoming document
        return supplierOrderRcvr.receive(supplierOrder);
    }
}
```

Code Example 4.3 Receiving an XML Document Through a Web Service Endpoint

JAX-RPC passes XML documents that are attachments to SOAP messages as abstract `Source` objects. Thus, you should assume no specific implementation—`StreamSource`, `SAXSource`, or `DOMSource`—for an incoming document. You should also not assume that the underlying JAX-RPC implementation will validate or parse the document before passing it to the Web service endpoint. The developer should programmatically ensure that the document is valid and conforms to an expected schema. (See the next section for more information about validation.) The developer should also ensure that the optimal API is used to bridge between the specific `Source` implementation passed to the endpoint and the intended processing model. See “Use the Most Appropriate API” on page 184.

Producing XML documents that are to be passed as attachments to SOAP operations can use any XML processing model, provided the resulting document can be wrapped into a `Source` object. The underlying JAX-RPC is in charge of attaching the passed document to the SOAP response message. For example, Code Example 4.4 and Code Example 4.5 show how to send and receive XML documents through a JMS queue.

```
public class SupplierOrderRcvr {
    private QueueConnectionFactory queueFactory;
```

```
    private Queue queue;

    public SupplierOrderRcvr() throws RemoteException {
        queueFactory = ...; // Lookup queue factory
        queue = ...; // Lookup queue
        ...
    }

    public String receive(Source supplierOrder)
        throws InvalidOrderException {
        // Preprocess (validate and transform) the incoming document
        String document = ...
        // Extract the order id from the incoming document
        String orderId = ...
        // Forward the transformed document to the processing layer
        // using JMS
        QueueConnection connection
            = queueFactory.createQueueConnection();
        QueueSession session = connection.createQueueSession(...);
        QueueSender queueSender = session.createSender(queue);
        TextMessage message = session.createTextMessage();
        message.setText(document);
        queueSender.send(message);
        return orderId;
    }
}
```

Code Example 4.4 Sending an XML Document to a JMS Queue

```
public class SupplierOrderMDB
        implements MessageDrivenBean, MessageListener {
    private OrderFulfillmentFacadeLocal poProcessor = null;

    public SupplierOrderMDB() {}

    public void ejbCreate() {
        // Create a purchase order processor
        poProcessor = ...
    }
```

```
    // Receives the supplier purchase order document from the
    // Web service endpoint (interaction layer) through a JMS queue
    public void onMessage(Message msg) {
        String document = ((TextMessage) msg).getText();
        // Processes the XML purchase order received by the supplier
        String invoice = poProcessor.processPO(document);
        ...
    }
}
```

Code Example 4.5 Receiving an XML Document Through a JMS Queue

There are circumstances when a Web service may internally exchange XML documents through a JMS queue or topic. When implementing an asynchronous architecture, the interaction layer of a Web service may send XML documents asynchronously using JMS to the processing layer. Similarly, when a Web service implements a workflow, the components implementing the individual stages of the workflow may exchange XML documents using JMS. From a developer's point of view, receiving or sending XML documents through a JMS queue or topic is similar in principle to the case of passing documents as SOAP message attachments. XML documents can be passed through a JMS queue or topic as text messages or in a Java-serialized form when those documents can be bound to Java objects.

4.3.3 Validating XML Documents

Once a document has been received or produced, a developer may—and most of the time must—validate the document against the schema to which it is supposed to conform. Validation, an important step in XML document handling, may be required to guarantee the reliability of an XML application. An application may legitimately rely on the parser to do the validation and thus avoid performing such validation itself.

However, because of the limited capabilities of some schema languages, a valid XML document may still be invalid in the application's domain. This might happen, for example, when a document is validated using DTD, because this schema language lacks capabilities to express strongly-typed data, complex unicity, and cross-reference constraints. Other modern schema languages, such as XSD, more rigorously—while still lacking some business constraint expressive-

ness—narrow the set of valid document instances to those that the business logic can effectively process. Regardless of the schema language, even when performing XML validation, the application is responsible for enforcing any uncovered domain-specific constraints that the document may nevertheless violate. That is, the application may have to perform its own business logic-specific validation in addition to the XML validation.

To decide where and when to validate documents, you may take into account certain considerations. Assuming a system—by system we mean a set of applications that compose a solution and that define a boundary within which trusted components can exchange information—one can enforce validation according to the following observations. (See Figure 4.3.)

1. Documents exchanged within the components of the system may not require validation.
2. Documents coming from outside the system, especially when they do not originate from external trusted sources, must be validated on entry.
3. Documents coming from outside the system, once validated, may be exchanged freely between internal components without further validation.

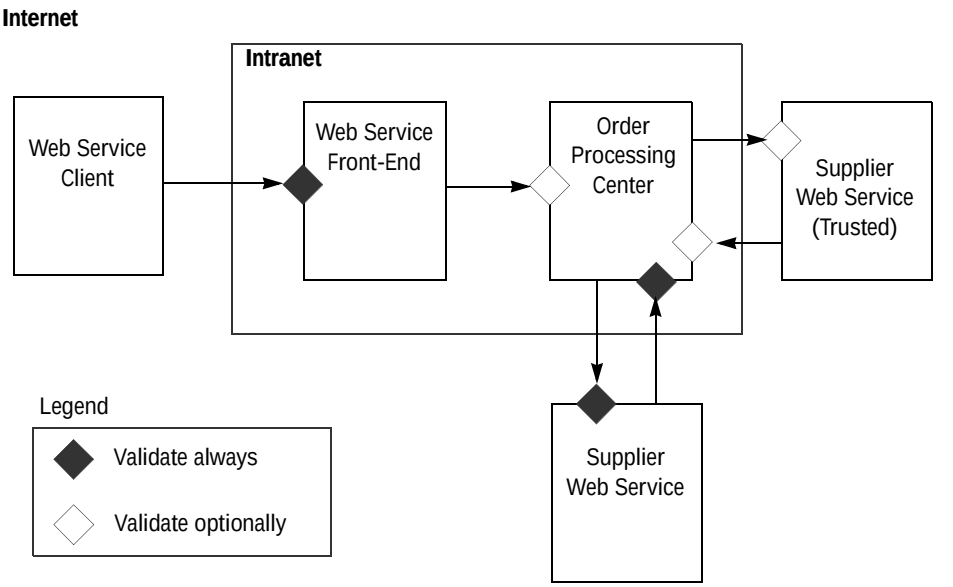

Figure 4.3 Validation of Incoming Documents

For example, a multitier e-business application that exchanges documents with trading partners through a front end enforces document validity at the front end. Not only does it check the validity of the document against its schema, but the application also ensures that the document type is a schema type that it can accept. It then may route documents to other applications or servers so that the proper services can handle them. Since they have already been validated, the documents do not require further validation. In a Web service, validation of incoming documents is typically performed in the interaction layer. Therefore, the processing layer may not have to validate documents it receives from the interaction layer.

Some applications may have to receive documents that conform to different schemas or different versions of a schema. In these cases, the application cannot do the validation up front against a specific schema unless the application is given a directive within the request itself about which schema to use. If no directive is included in the request, then the application has to rely on a hint provided by the document itself. Note that to deal with successive versioning of the same schema—where the versions actually modify the overall application's interface—it sometimes may be more convenient for an application to expose a separate endpoint for each version of the schema.

To illustrate, an application must check that the document is validated against the expected schema, which is not necessarily the one to which the document declares it conforms. With DTD schemas, this checking can be done only after validation. When using DOM, the application may retrieve the system or public identifier (`SystemID` or `PublicID`) of the DTD to ensure it is the identifier of the schema expected (Code Example 4.6), while when using SAX, it can be done on the fly by handling the proper event. With JAXP 1.2 and XSD (or other non-DTD schema languages), the application can specify up-front the schema to validate against (Code Example 4.7); the application can even ignore the schema referred to by the document itself.

```
public static boolean checkDocumentType(Document document,
        String dtdPublicId) {
    DocumentType documentType = document.getDoctype();
    if (documentType != null) {
        String publicId = documentType.getPublicId();
        return publicId != null && publicId.equals(dtdPublicId);
    }
```

```
    return false;
}
```

Code Example 4.6 Ensuring the Expected Type of a DTD-Conforming Document

```
public static final String W3C_XML_SCHEMA
    = "http://www.w3.org/2001/XMLSchema";
public static final String JAXP_SCHEMA_LANGUAGE
    = "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
public static final String JAXP_SCHEMA_SOURCE
    = "http://java.sun.com/xml/jaxp/properties/schemaSource";

public static SAXParser createParser(boolean validating,
        boolean xsdSupport, CustomEntityResolver entityResolver,
        String schemaURI) throws ... {
    // Obtain a SAX parser from a SAX parser factory
    SAXParserFactory parserFactory
        = SAXParserFactory.newInstance();
    // Enable validation
    parserFactory.setValidating(validating);
    parserFactory.setNamespaceAware(true);
    SAXParser parser = parserFactory.newSAXParser();
    if (xsdSupport) { // XML Schema Support
        try {
            // Enable XML Schema validation
            parser.setProperty(JAXP_SCHEMA_LANGUAGE,
                W3C_XML_SCHEMA);
            // Set the validating schema to the resolved schema URI
            parser.setProperty(JAXP_SCHEMA_SOURCE,
                entityResolver.mapEntityURI(schemaURI));
        } catch(SAXNotRecognizedException exception) { ... }
    }
    return parser;
}
```

Code Example 4.7 Setting the Parser for Validation in JAXP 1.2

When relying on the schemas to which documents internally declare they are conforming (through a DTD declaration or an XSD hint), for security and to avoid external malicious modification, you should keep your own copy of the schemas and validate against these copies. This can be done using an entity resolver, which is an interface from the SAX API (`org.xml.sax.EntityResolver`), that forcefully maps references to well-known external schemas to secured copies.

To summarize these recommendations:

- ❒ Validate incoming documents at the system boundary, especially when documents come from untrusted sources.
- ❒ When possible, enforce validation up-front against the supported schemas.
- ❒ When relying on internal schema declarations (DTD declaration, XSD hint, and so forth):
  - ❒ Reroute external schema references to secured copies.
  - ❒ Check that the validating schemas are supported schemas.

4.3.4 Mapping Schemas to the Application Data Model

After defining the application interface and the schemas of the documents to be consumed and produced, the developer has to define how the document schemas relate or map to the data model on which the application applies its business logic. We refer to these document schemas as external schemas. These schemas may be specifically designed to meet the application's requirements, such as when no preexisting schemas are available, or they may be imposed on the developer. The latter situation, for example, may occur when the application intends to be part of an interacting group within an industry promoting standard vertical schemas. (For example, UBL or ebXML schemas.)

4.3.4.1 Mapping Design Strategies

Depending on an application's requirements, there are three main design strategies or approaches for mapping schemas to the application data model. (See Figure 4.4.)

1. An "out-to-in" approach—The developer designs the internal data model based on the external schemas.
2. A "meet-in-the-middle" approach—The developer designs the data model along with an internal generic matching schema. Afterwards, the developer de-

fines transformations on the internal schema to support the external schemas.

3. An "in-to-out" approach, or legacy adapter—This approach is actually about how to map an application data model to schemas. The developer designs the exposed schema from an existing data model.

Figure 4.4, Figure 4.5, and Figure 4.6 show the sequencing of the activities involved at design time and the artifacts (schemas and classes) used or produced by these activities. The figures also show the relationships between these artifacts and the runtime entities (documents and objects), as well as the interaction at runtime between these entities.

The first approach (Figure 4.4), which introduces a strong dependency between the application's data model and logic and the external schemas, is suitable only for applications dedicated to supporting a specific interaction model. A strong dependency such as this implies that evolving or revising the external schemas impacts the application's data model and its logic.

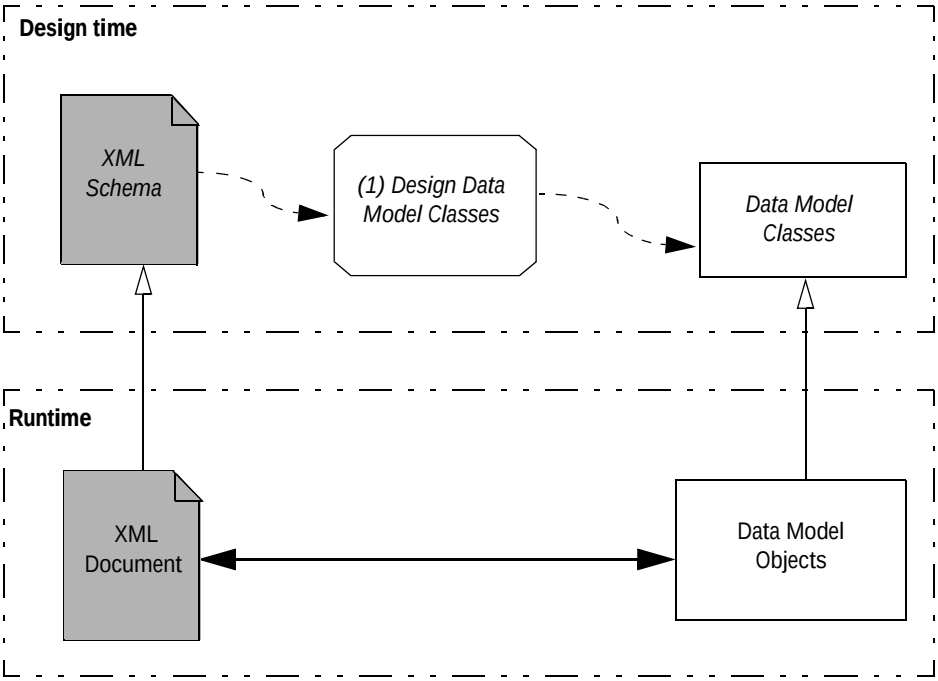

Figure 4.4 Out-to-In Approach for Mapping Schemas to the Data Model Classes

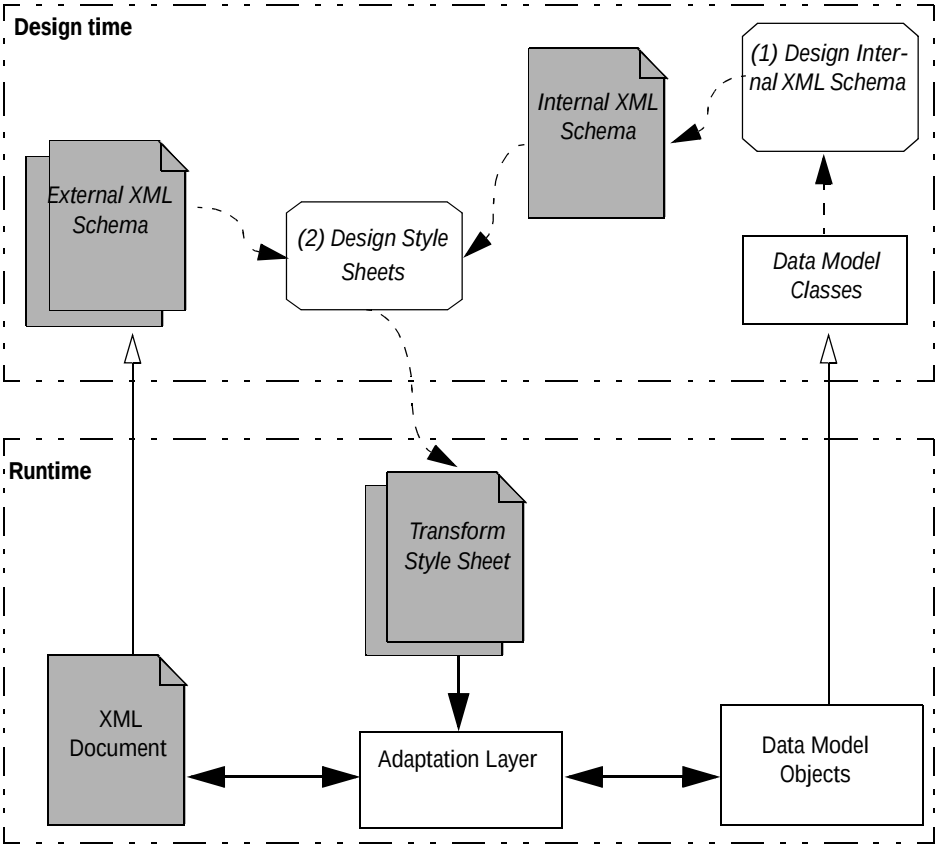

Figure 4.5 Meet-in-the-Middle Approach for Mapping Schemas to Data Model Classes

The second approach (Figure 4.5), which introduces a transformation or adaptation layer between the application's data model and logic and the external schemas, is particularly suitable for applications that may have to support different external schemas. Having a transformation layer leaves room for supporting additional schemas and is a natural way to account for the evolution of a particular schema. The challenge is to devise an internal schema that is sufficiently generic and that not only shields the application from external changes (in number and revision) but also allows the application to fully operate and interoperate. Typically, such an internal schema either maps to a minimal operational subset or common denominator of the external schemas, or it maps to a generic, universal

schema of the application's domain. (UBL is an example of the latter case.) The developer must realize that such an approach has some limitations—it is easier to transform from a structure containing more information to one with less information than the reverse. Therefore, the choice of the generic internal schema is key to that approach. Code Example 4.8 shows how to use a stylesheet to transform an external, XSD-based schema to an internal, DTD-based schema.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:so="http://blueprints.j2ee.sun.com/SupplierOrder"
    xmlns:li="http://blueprints.j2ee.sun.com/LineItem"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="1.0">

    <xsl:output method="xml" indent="yes" encoding="UTF-8"
        doctype-public="-//Sun Microsystems, Inc. -
            J2EE Blueprints Group//DTD SupplierOrder 1.1//EN"
        doctype-system="/com/sun/j2ee/blueprints/
            supplierpo/rsrc/schemas/SupplierOrder.dtd" />

    <xsl:template match="/">
        <SupplierOrder>
            <OrderId><xsl:value-of select="/
            so:SupplierOrder/so:OrderId" /></OrderId>
            <OrderDate><xsl:value-of select="/
                so:SupplierOrder/so:OrderDate" /></OrderDate>
            <xsl:apply-templates select=".//
            so:ShippingAddress|.//li:LineItem"/>
        </SupplierOrder>
    </xsl:template>

    <xsl:template match="/so:SupplierOrder/
            so:ShippingAddress">
        ...
    </xsl:template>

    <xsl:template match="/so:SupplierOrder/so:LineItems/
            li:LineItem">
```

```
            ...
        </xsl:template>
    </xsl:stylesheet>
```

Code Example 4.8 Stylesheet for Transforming from External XSD-Based Schema to Internal DTD-Based Schema

Normally, developers should begin this design process starting from the application's interface definition plus the XML schemas. In some situations, a developer may have to work in reverse; that is, start from the inside and work out. See the third approach, shown in Figure 4.6.

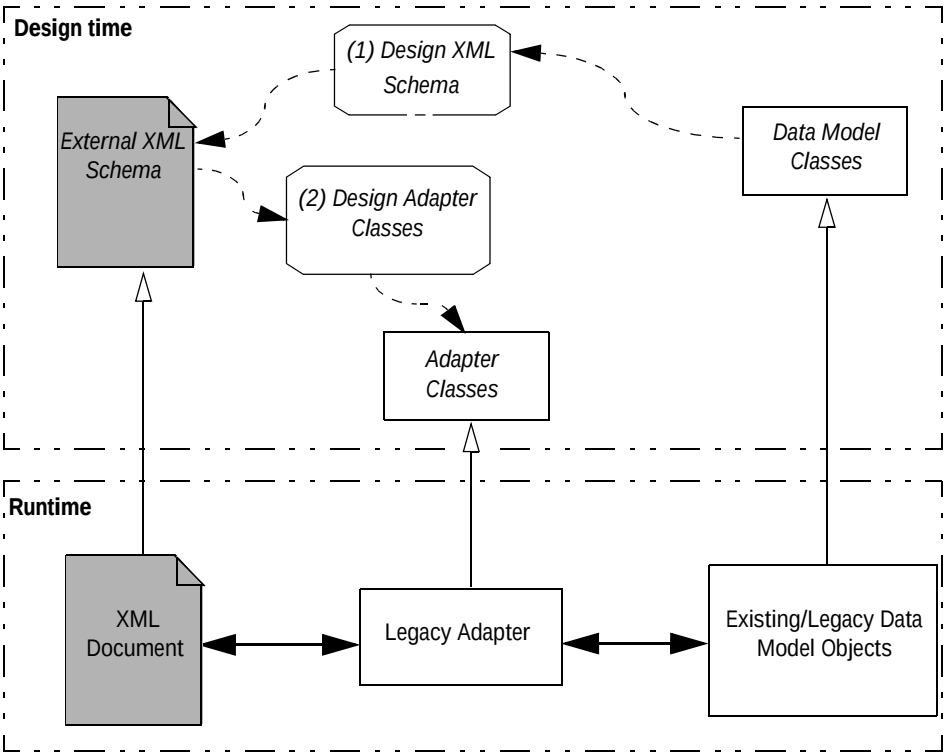

Figure 4.6 Legacy Adapter (In-to-Out) Approach for Mapping Schemas to Data Model Classes

The developer may have to use the application's data model to create a set of matching schemas, which would then be exposed as part of the application's interface. This third approach is often used when existing or legacy applications need to expose an XML-based interface to facilitate a loosely coupled integration in a broader enterprise system. This technique is also known as legacy adapters or wrappers. In these cases, the application's implementation determines the interface and the schemas to expose. In addition, this approach can be combined with the meet-in-the-middle approach to provide a proper adaptation layer, which in turn makes available a more interoperable interface that is, an interface that is not so tightly bound to the legacy application. See Chapter 6 for more information on application integration.

4.3.4.2 Flexible Mapping

In complement to these approaches, it is possible to map complete documents or map just portions of documents. Rather than a centralized design for mapping from an external schema to a well-defined internal schema, developers can use a decentralized design where components map specific portions of a document to an adequate per-component internal representation. Different components may require different representations of the XML documents they consume or produce. Such a decentralized design allows for flexible mapping where:

- A component may not need to know the complete XML document. A component may be coded against just a fragment of the overall document schema.
- The document itself may be the persistent core representation of the data model. Each component maps only portions of the document to transient representation in order to apply their respective logic and then modifies the document accordingly.
- Even if the processing model is globally document-centric (see "Choosing Processing Models" on page 151), each component can—if adequate—locally implement a more object-centric processing model by mapping portions of the document to domain-specific objects.
- Each component can handle the document using the most effective or suitable XML processing technique. (See "Choosing an XML Processing Programming Model" on page 164.)

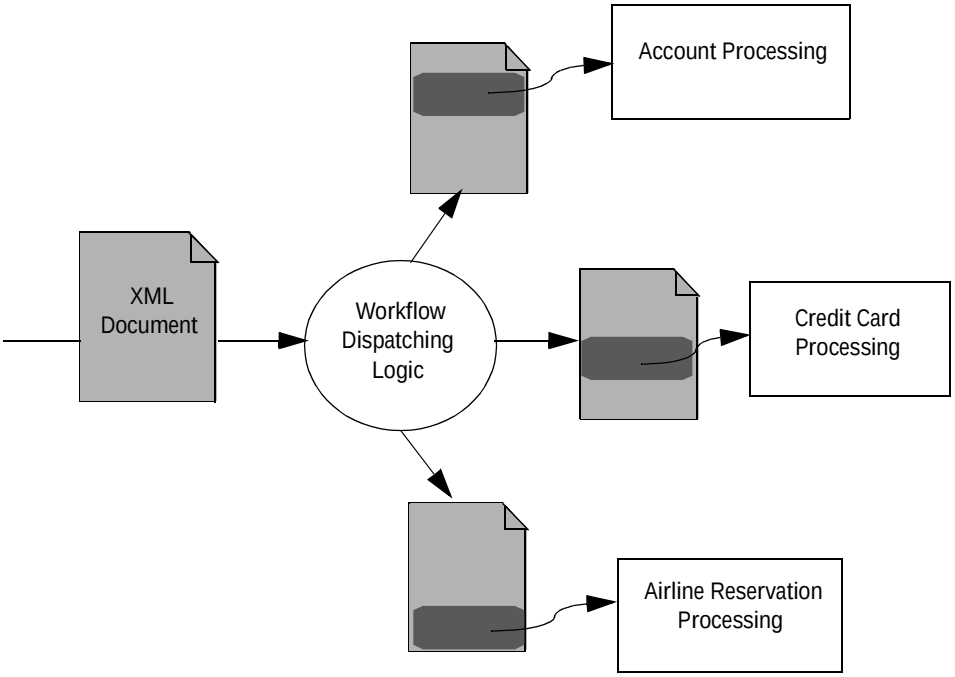

Figure 4.7 Flexible Mapping Applied to a Travel Service Scenario

This technique is particularly useful for implementing a document-oriented workflow where components exchange or have access to entire documents but only manipulate portions of the documents. For example, Figure 4.7 shows how a `PurchaseOrder` document may sequentially go through all the stages of a workflow. Each stage may process specific information within the document. A credit card processing stage may only retrieve the `CreditCard` element from the `PurchaseOrder` document. Upon completion, a stage may "stamp" the document by inserting information back into the document. In the case of a credit card processing stage, the credit card authorization date and status may be inserted back into the `PurchaseOrder` document.

4.3.4.3 XML Componentization

For a document-centric processing model, especially when processing documents in the EJB tier, you may want to create generic, reusable components whose state is serializable to and from XML. (See "Designing Domain-Specific XML Schemas"

on page 131.) For example, suppose your application works with an `Address` entity bean whose instances are initialized with information retrieved from various XML documents, such as purchase order and invoice documents. Although the XML documents conform to different schemas, you want to use the same component—the same `Address` bean—without modification regardless of the underlying supported schema.

A good way to address this issue is to design a generic XML schema into which your component state can be serialized. From this generic schema, you can generate XML-serializable domain-specific or content objects that handle the serialization of your component state.

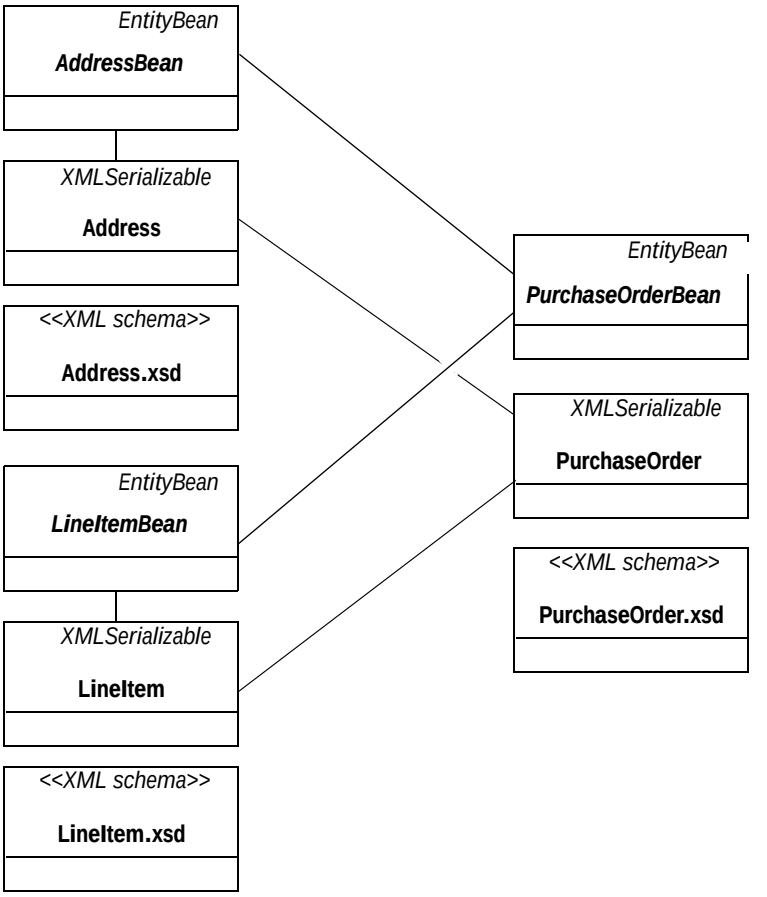

Figure 4.8 Composite XML-Serializable Component

You can generate the content objects manually or automatically by using XML data-binding technologies such as JAXB. Furthermore, you can combine these XML-serializable components into composite entities with corresponding composite schemas.

When combined with the "meet-in-the-middle" approach discussed previously, you can apply XSLT transformations to convert XML documents conforming to external vertical schemas into your application's supported internal generic schemas. Transformations can also be applied in the opposite direction to convert documents from internal generic schemas to external vertical schemas.

For example, Figure 4.8, which illustrates XML componentization, shows a `PurchaseOrderBean` composite entity and its two components, `AddressBean` and `LineItemBean`. The schemas for the components are composed in the same way and form a generic composite schema. Transformations can be applied to convert the internal generic composite `PurchaseOrder` schema to and from several external vertical schemas. Supporting an additional external schema is then just a matter of creating a new stylesheet.

4.3.5 Choosing Processing Models

An XML-based application may either apply its business logic directly on consumed or produced documents, or it may apply its logic on domain-specific objects that completely or partially encapsulate the content of such documents. Domain-specific objects are Java objects that may not only encapsulate application domain-specific data, but may also embody application domain-specific behavior.

An application's business logic may directly handle documents it consumes or produces, which is called a document-centric processing model, if the logic:

- Relies on both document content and structure
- Is required to punctually modify incoming documents while preserving most of their original form, including comments, external entity references, and so forth

In a document-centric processing model, the document processing may be entangled with the business logic and may therefore introduce strong dependencies between the business logic and the schemas of the consumed and produced documents—the "meet-in-the-middle" approach (discussed in "Mapping Schemas to the Application Data Model" on page 143) may, however, alleviate this problem. Moreover, the document-centric processing model does not promote a clean

separation between business and XML programming skills, especially when an application developer who is more focused on the implementation of the business logic must additionally master one or several of the XML processing APIs.

There are cases that require a document-centric processing model, such as:

- The schema of the processed documents is only partially known and therefore cannot be completely bound to domain-specific objects; the application edits only the known part of the documents before forwarding them for further processing.
- Because the schemas of the processed documents may vary or change greatly, it is not possible to hard-code or generate the binding of the documents to domain-specific objects; a more flexible solution is required, such as one using DOM with XPath.

A typical document-centric example is an application that implements a data-driven workflow: Each stage of the workflow processes only specific information from the incoming document contents, and there is no central representation of the content of the incoming documents. A stage of the workflow may receive a document from an earlier stage, extract information from the document, apply some business logic on the extracted information, and potentially modify the document before sending it to the next stage.

Generally, it is best to have the application's business logic directly handle documents only in exceptional situations, and to do so with great care. You should instead consider applying the application's business logic on domain-specific objects that completely or partially encapsulate the content of consumed or produced documents. This helps to isolate the business logic from the details of XML processing.

Keep in mind that schema-derived classes, which are generated by JAXB and other XML data-binding technologies (see "XML Data-Binding Programming Model" on page 169), usually completely encapsulate the content of a document. While these schema-derived classes isolate the business logic from the XML processing details—specifically parsing, validating, and building XML documents—they still introduce strong dependencies between the business logic and the schemas of the consumed and produced documents. Because of these strong dependencies, and because they may still retain some document-centric characteristics (especially constraints), applications may still be considered document centric when they apply business logic directly on classes generated by XML

data-binding technologies from the schemas of consumed and produced documents. To change to a pure object-centric model, the developer may move the dependencies on the schemas down by mapping schema-derived objects to domain-specific objects. The domain-specific object classes expose a constant, consistent interface to the business logic but internally delegate the XML processing details to the schema-derived classes. Overall, such a technique reduces the coupling between the business logic and the schema of the processed documents. "Abstracting XML Processing from Application Logic" on page 155 discusses a generic technique for decoupling the business logic and the schema of the processed documents.

A pure object-centric processing model requires XML-related issues to be kept at the periphery of an application—that is, in the Web service interaction layer closest to the service endpoint, or, for more classical applications, in the Web tier. In this case, XML serves only as an additional presentation media for the application's inputs and outputs. When implementing a document-oriented workflow in the processing layer of a Web service, or when implementing the asynchronous Web service interaction layer presented in "Delegating Web Service Requests to Processing Layer" on page 92, an object-centric processing model may still be enforced by keeping the XML-related issues within the message-driven beans that exchange documents.

Note that the object- and document-centric processing models may not be exclusive of one another. When using the flexible mapping technique mentioned earlier, an application may be globally document-centric and exchange documents between its components, and some components may themselves locally process part of the documents using an object-centric processing model. Each component may use the most adequate processing model for performing its function.

4.3.6 Fragmenting Incoming XML Documents

When your service's business logic operates on the contents of an incoming XML document, it is a good idea to break XML documents into logical fragments when appropriate. The processing logic receives an XML document containing all information for processing a request. However, the XML document usually has well-defined segments for different entities, and each segment contains the details about a specific entity.

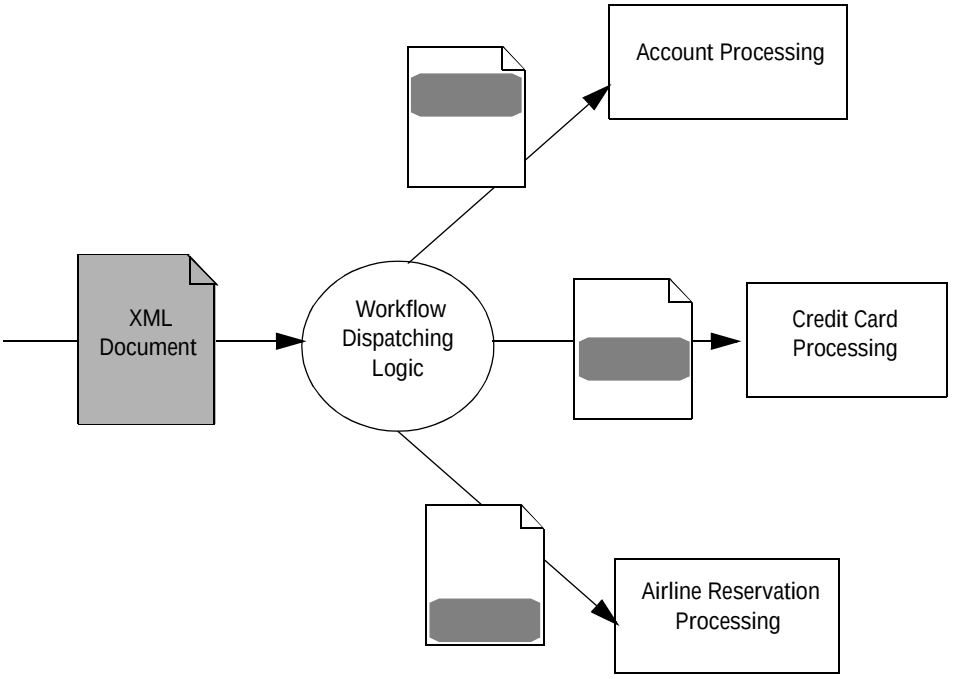

Figure 4.9 Fragmenting an Incoming XML Document for a Travel Service

- Rather than pass the entire document to different components handling various stages of the business process, it's best if the processing logic breaks the document into fragments and passes only the required fragments to other components or services that implement portions of the business process logic.

Figure 4.9 shows how the processing layer might process an XML document representing an incoming purchase order for a travel agency Web service. The document contains details such as account information, credit card data, travel destinations, dates, and so forth. The business logic involves verifying the account, authorizing the credit card, and filling the airline and hotel portions of the purchase order. It is not necessary to pass all the document details to a business process stage that is only performing one piece of the business process, such as account verification. Passing the entire XML document to all stages of the business process results in unnecessary information flows and extra processing. It is more efficient to extract the logical fragments—account fragment, credit card

fragment, and so forth—from the incoming XML document and then pass these individual fragments to the appropriate business process stages in an appropriate format (DOM tree, Java object, serialized XML, and so forth) expected by the receiver.

While it is complementary to most of the mapping design strategies presented in "Mapping Design Strategies" on page 143, this technique is best compared against the flexible mapping design strategy. (See "Flexible Mapping" on page 148.) Flexible mapping advocates a decentralized mapping approach: Components or stages in a workflow each handle the complete incoming document, but each stage only processes the appropriate part of the document. Fragmenting an incoming document can be viewed as a centralized implementation of the flexible mapping design. Fragmenting an incoming document, by suppressing redundant parsing of the incoming document and limiting the exchanges between stages to the strictly relevant data, improves performance over a straightforward implementation of flexible mapping. However, it loses some flexibility because the workflow dispatching logic is required to specifically know about (and therefore depend on) the document fragments and formats expected by the different stages.

Fragmenting a document has the following benefits:

- It avoids extra processing and exchange of superfluous information throughout the workflow.
- It maximizes privacy because it limits sending sensitive information through the workflow.
- It centralizes some of the XML processing tasks of the workflow and therefore simplifies the overall implementation of the workflow.
- It provides greater flexibility to workflow error handling since each stage handles only business logic-related errors while the workflow dispatching logic handles document parsing and validation errors.

4.3.7 Abstracting XML Processing from Application Logic

As mentioned earlier, the developer of an XML-based application and more specifically of a Web service application, may have to explicitly handle XML in the following layers of the application:

- In the interaction layer of a Web service in order to apply some pre- or post-processing, such as XML validation and transformation to the exchanged doc-

uments. (See “Receiving Requests” on page 89 and “Formulating Responses” on page 98.) Moreover, when the processing layer of a Web service implements an object-centric processing model, the interaction layer may be required to map XML documents to or from domain-specific objects before delegating to the processing layer by using one of the three approaches for mapping XML schemas to the application data model presented in “Mapping Schemas to the Application Data Model” on page 143.

- In the processing layer of a Web service when implementing a document-centric processing model. (See “Handling XML Documents in a Web Service” on page 105.) In such a case, the processing layer may use techniques such as flexible mapping or XML componentization. See “Flexible Mapping” on page 148 and “XML Componentization” on page 149.

With the object-centric processing model—when XML document content is mapped to domain-specific objects—the application applies its business logic on the domain-specific objects rather than the documents. In this case, only the interaction logic may handle documents. However, in the document-centric model, the application business logic itself may directly have to handle the documents. In other words, some aspects of the business model may be expressed in terms of the documents to be handled.

There are drawbacks to expressing the business model in terms of the documents to be handled. Doing so may clutter the business logic with document processing-related logic, which should be hidden from application developers who are more focused on the implementation of the business logic. It also introduces strong dependencies between the document’s schemas and the business logic, and this may cause maintainability problems particularly when handling additional schemas or supporting new versions of an original schema (even though those are only internal schemas to which documents originally conforming to external schemas have been converted). Additionally, since there are a variety of APIs that support various XML processing models, such a design may lock the developer into one particular XML-processing API. It may make it difficult, and ineffective from a performance perspective, to integrate components that use disparate processing models or APIs.

The same concerns—about maintainability in the face of evolution and the variety of XML processing models or APIs—apply to some extent for the logic of the Web service interaction layer, which may be in charge of validating exchanged

documents, transforming them from external schemas to internal schemas and, in some cases, mapping them to domain-specific objects.

For example, consider a system processing a purchase order that sends the order to a supplier warehouse. The supplier, to process the order, may need to translate the incoming purchase order from the external, agreed-upon schema (such as an XSD-based schema) to a different, internal purchase order schema (such as a DTD-based schema) supported by its components. Additionally, the supplier may want to map the purchase order document to a purchase order business object. The business logic handling the incoming purchase order must use an XML-processing API to extract the information from the document and map it to the purchase order entity. In such a case, the business logic may be mixed with the document-handling logic. If the external purchase order schema evolves or if an additional purchase order schema needs to be supported, the business logic will be impacted. Similarly, if for performance reasons you are required to revisit your choice of the XML-processing API, the business logic will also be impacted. The initial choice of XML-processing API may handicap the integration of other components that need to retrieve part or all of the purchase order document from the purchase order entity.

The design shown in Figure 4.10, which we refer to as the XML document editor (XDE) design, separates application logic (business or interaction logic) from document processing logic. Following a design such as this helps avoid the problems just described.

The term "Editor" used here refers to the capability to programmatically create, access, and modify—that is, edit—XML documents. The XML document editor design is similar to the data access object design strategy, which abstracts database access code from a bean's business logic.

The XML document editor implements the XML document processing using the most relevant API, but exposes only methods relevant to the application logic. Additionally, the XML document editor should provide methods to set or get documents to be processed, but should not expose the underlying XML processing API. These methods should use the abstract `Source` class (and `Result` class) from the JAXP API, in a similar fashion as JAX-RPC, to ensure that the underlying XML-processing API remains hidden. If requirements change, you can easily switch to a different XML processing technique without modifying the application logic. Also, a business object (such as an enterprise bean) that processes XML documents through an XML document editor should itself only expose accessor methods that use the JAXP abstract `Source` or `Result` class. Moreover, a business object or a service endpoint can use different XML document editor

design strategies, combined with other strategies for creating factory methods or abstract factories (strategies for creating new objects where the instantiation of those objects is deferred to a subclass), to uniformly manipulate documents that conform to different schemas. The business object can invoke a factory class to create instances of different XML document editor implementations depending on the schema of the processed document. This is an alternate approach to applying transformations for supporting several external schemas.

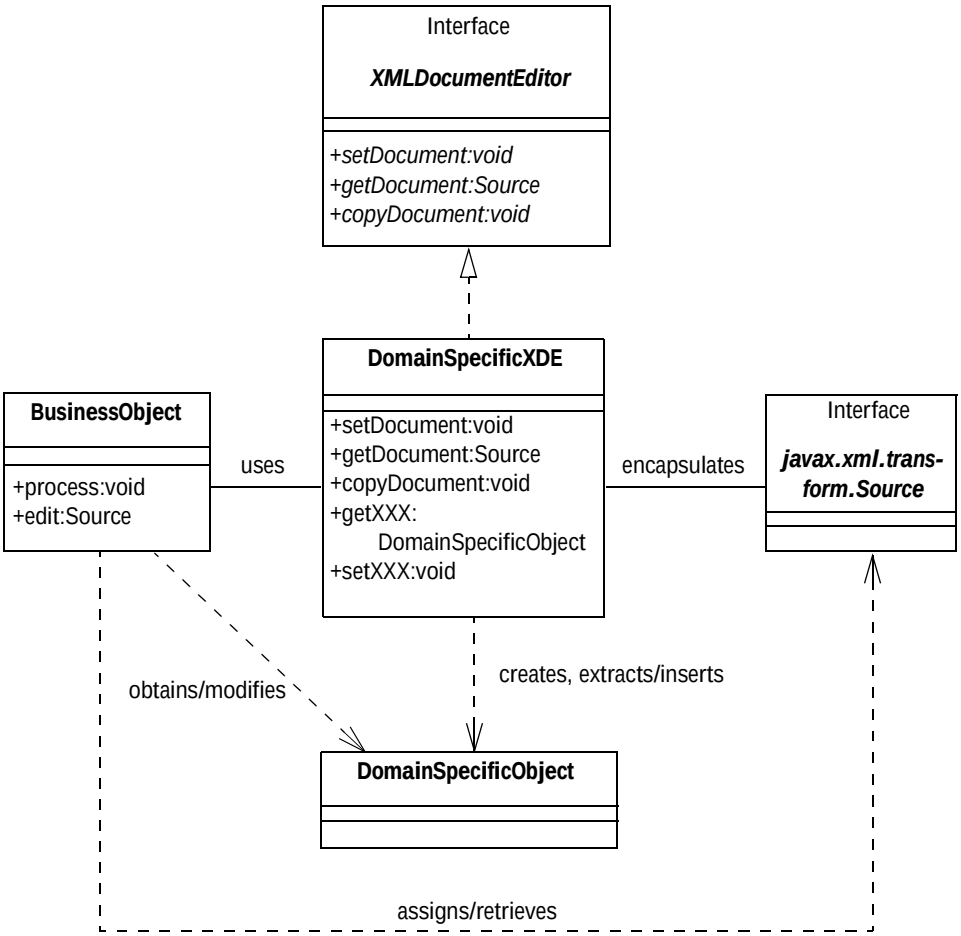

Figure 4.10 Basic Design of an XML Document Editor

Figure 4.10 shows the class diagram for a basic XML document editor design, while Figure 4.11 shows the class diagram for an XML document editor factory design. You should consider using a similar design in the following situations:

- When you want to keep the business objects focused on business logic and keep code to interact with XML documents separate from business logic code.

- In a similar way, when you want to keep the Web service endpoints focused on interaction logic and keep code to pre- and post-process XML documents separate from interaction logic code.

Figure 4.11 Factory Design to Create Schema-Specific XML Document Editors

- When you want to implement a flexible mapping design where each component may manipulate a common document in the most suitable manner for itself.
- When requirements might evolve (such as a new schema to be supported or a new version of the same schema) to where they would necessitate changes to the XML-processing implementation. Generally, you do not want to alter the application logic to accommodate these XML-processing changes. Additionally, since several XML-processing APIs (SAX, DOM, XSLT, JAXB technology, and so forth) may be relevant, you want to allow for subsequent changes to later address such issues as performance and integration.
- When different developer skill sets exist. For example, you may want the business domain and XML-processing experts to work independently. Or, you may want to leverage particular skill sets within XML-processing techniques.

Figure 4.12 and Code Example 4.9 give an example of a supplier Web service endpoint using an XML document editor to preprocess incoming purchase order documents.

```
public class SupplierOrderXDE extends
          XMLDocumentEditor.DefaultXDE {
    public static final String DEFAULT_ENCODING = "UTF-8";
    private Source source = null;
    private String orderId = null;

    public SupplierOrderXDE(boolean validating, ...) {
        // Initialize XML processing logic
    }
    // Sets the document to be processed
    public void setDocument(Source source) throws ... {
        this.source = source;
    }
    // Invokes XML processing logic to validate the source document,
    // extract its orderId, transform it into a different format,
    // and copy the resulting document into the Result object
    public void copyDocument(Result result) throws ... {
        orderId = null;
        // XML processing...
    }
```

```
    // Returns the processed document as a Source object
    public Source getDocument() throws ... {
        return new StreamSource(new StringReader(
            getDocumentAsString()));
    }
    // Returns the processed document as a String object
    public String getDocumentAsString() throws ... {
        ByteArrayOutputStream stream = new ByteArrayOutputStream();
        copyDocument(new StreamResult(stream));
        return stream.toString(DEFAULT_ENCODING);
    }
    // Returns the orderId value extracted from the source document
    public String getOrderId() {
        return orderId;
    }
}
```

Code Example 4.9 Supplier Service Endpoint Using XML Document Editor

❒ To summarize, it is recommended that you use a design similar to the XML Document Editor presented above to abstract and encapsulate all XML document processing. In turn, the business object or service endpoint using such a document editor only invokes the simple API provided by the document editor. This hides all the complexities and details of interacting with the XML document from the business object clients.

As noted earlier, this design is not limited to the document-centric processing model where the application applies its business logic on the document itself. In an object-centric processing model, document editors can be used by the Web service interaction layer closest to the service endpoint, to validate, transform, and map documents to or from domain-specific objects. In this case, using the document editor isolates the interaction logic from the XML processing logic.

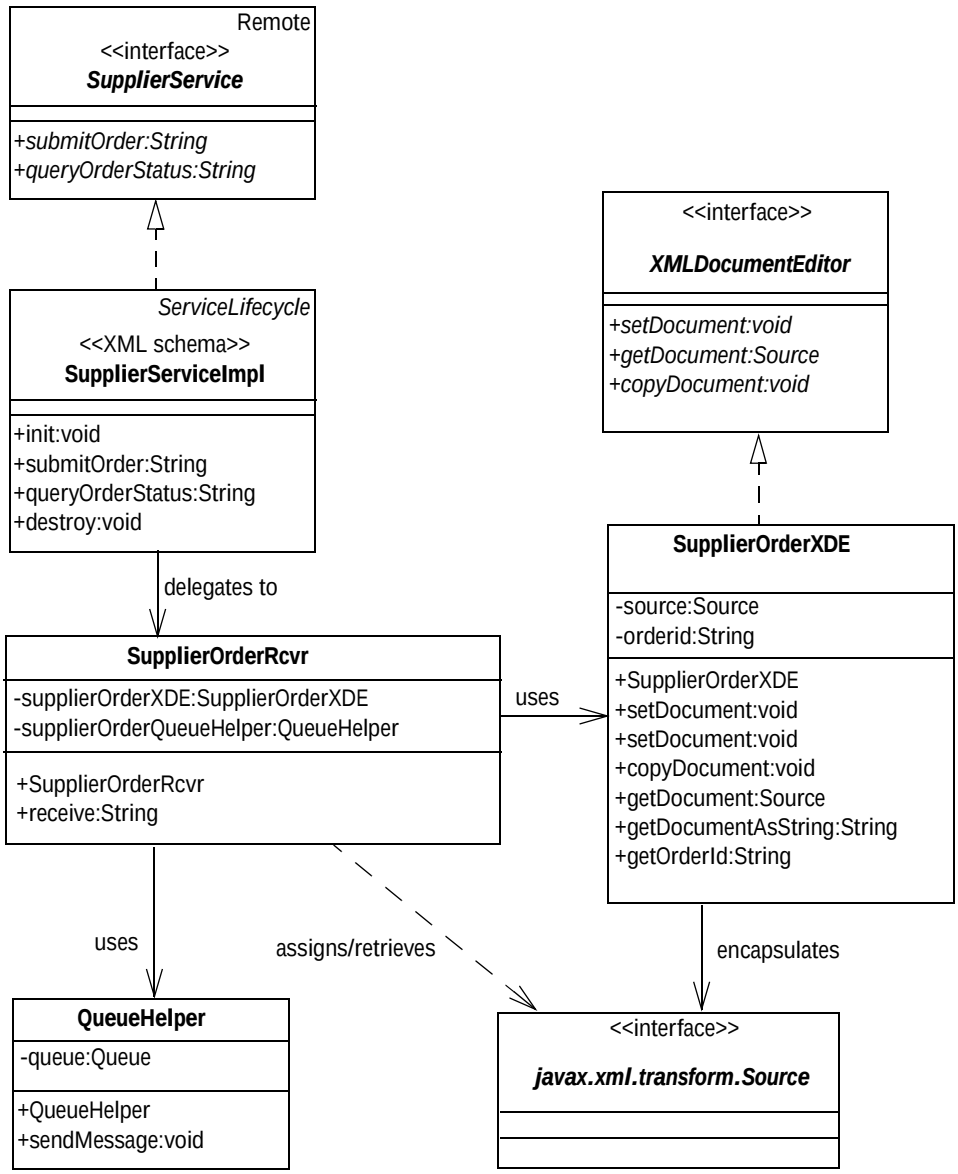

Figure 4.12 Class Diagram of Supplier Service Using XML Document Editor

4.3.8 Design Recommendation Summary

When you design an XML-based application, specifically one that is a Web service, you must make certain decisions concerning the processing of the content of incoming XML documents. Essentially, you decide the "how, where, and what" of the processing: You decide the technology to use for this process, where to perform the processing, and the form of the content of the processing.

In summary, keep in mind the following recommendations:

- ❒ When designing application-specific schemas, promote reuse, modularization, and extensibility, and leverage existing vertical and horizontal schemas.
- ❒ When implementing a pure object-centric processing model, keep XML on the boundary of your system as much as possible—that is, in the Web service interaction layer closest to the service endpoint, or, for more classical applications, in the presentation layer. Map document content to domain-specific objects as soon as possible.
- ❒ When implementing a document-centric processing model, consider using the flexible mapping technique. This technique allows the different components of your application to handle XML in a way that is most appropriate for each of them.
- ❒ Strongly consider validation at system entry points of your processing model—specifically, validation of input documents where the source is not trusted.
- ❒ When consuming or producing documents, as much as possible express your documents in terms of abstract `Source` and `Result` objects that are independent from the actual XML-processing API you are using.
- ❒ Consider a "meet-in-the-middle" mapping design strategy when you want to decouple the application data model from the external schema that you want to support.
- ❒ Abstract XML processing from the business logic processing using the XML document editor design strategy. This promotes separation of skills and independence from the actual API used.

4.4 Implementing XML-Based Applications

You must make some decisions when formulating the implementation of an XML-based application. Briefly, you have to choose the XML programming models for your application. Note that multiple programming models are available and these models may be relevant for different situations—models may be complementary or even competing. As such, your application may use different models, sometimes even in conjunction with one another. That is, you may have to combine programming models in what may be called XML processing pipelines.

You may also have to consider and address other issues. For example, you may have to determine how to resolve external entity references in a uniform way across your application regardless of the programming models used.

4.4.1 Choosing an XML Processing Programming Model

A J2EE developer has the choice of four main XML processing models, available through the following APIs:

1. Simple API for XML Parsing (SAX), which provides an event-based programming model
2. Document Object Model (DOM), which provides an in-memory tree-traversal programming model
3. XML data-binding, which provides an in-memory Java content class-bound programming model
4. eXtensible Stylesheet Language Transformations (XSLT), which provides a template-based programming model

The most common processing models are SAX and DOM. These two models along with XSLT are available through the JAXP APIs. (See “Java™ APIs for XML Processing” on page 41.) The XML data binding model is available through the JAXB technology. (See “Emerging Standards” on page 40.)

Processing an XML document falls into two categories. Not only does it encompass parsing a source XML document so that the content is available in some form for an application to process, processing also entails writing or producing an XML document from content generated by an application. Parsing an XML representation into an equivalent data structure usable by an application is often called deserialization, or unmarshalling. Similarly, writing a data structure to an

equivalent XML representation is often called serialization or marshalling. Some processing models support both processing types, but others, such as SAX, do not.

- Just as you would avoid manually parsing XML documents, you should avoid manually constructing XML documents. It is better to rely on higher-level, reliable APIs (such as DOM, and DOM-like APIs, or JAXB technology) to construct XML documents, because these APIs enforce the construction of well-formed documents. In some instances, these APIs also allow you to validate constructed XML documents.

  Now let's take a closer look at these XML processing APIs.

4.4.1.1 SAX Programming Model

When you use SAX to process an XML document, you have to implement event handlers to handle events generated by the parser when it encounters the various tokens of the markup language. Because a SAX parser generates a transient flow of these events, it is advisable to process the source document in the following fashion. Intercept the relevant type of the events generated by the parser. You can use the information passed as parameters of the events to help identify the relevant information that needs to be extracted from the source document. Once extracted from the document, the application logic can process the information.

Typically, with SAX processing, an application may have to maintain some context so that it can logically aggregate or consolidate information from the flow of events. Such consolidation is often done before invoking or applying the application's logic. The developer has two choices when using SAX processing:

1. The application can "on the fly" invoke the business logic on the extracted information. That is, the logic is invoked as soon as the information is extracted or after only a minimal consolidation. With this approach, referred to as stream processing, the document can be processed in one step.
2. The application invokes the business logic after it completes parsing the document and has completely consolidated the extracted information. This approach takes two steps to process a document.

Note that what we refer to as consolidated information may in fact be domain-specific objects that can be directly passed to the business logic.

Stream processing (the first approach) lets an application immediately start processing the content of a source document. Not only does the application not have to wait for the entire document to be parsed, but, in some configurations, the application does not have to wait for the entire document to be retrieved. This includes retrieving the document from an earlier processing stage when implementing pipelines, or even retrieving the document from the network when exchanging documents between applications.

Stream processing, while it offers some performance advantages, also has some pitfalls and issues that must be considered. For instance, a document may appear to be well-formed and even valid for most of the processing. However, there may be unexpected errors by the end of the document that cause the document to be broken or invalid. An application using stream processing notices these problems only when it comes across erroneous tokens or when it cannot resolve an entity reference. Or, the application might realize the document is broken if the input stream from which it is reading the document unexpectedly closes, as with an end-of-file exception. Thus, an application that wants to implement a stream processing model may have to perform the document parsing and the application's business logic within the context of a transaction. Keeping these operations within a transaction leverages the container's transaction capabilities: The container's transaction mode accounts for unexpected parsing errors and rolls back any invalidated business logic processing.

With the second approach, parsing the document and applying business logic are performed in two separate steps. Before invoking the application's business logic, the application first ensures that the document and the information extracted from the document are valid. Once the document data is validated, the application invokes the business logic, which may be executed within a transaction if need be.

The SAX programming model provides no facility to produce XML documents. However, it is still possible to generate an XML document by initiating a properly balanced sequence of events—method calls—on a custom serialization handler. The handler intercepts the events and, using an XSLT identity transformation operation, writes the events in the corresponding XML syntax. The difficulty for the application developer lies in generating a properly balanced sequence of events. Keep in mind, though, that generating this sequence of events is prone to error and should be considered only for performance purposes.

SAX generally is very convenient for extracting information from an XML document. It is also very convenient for data mapping when the document structure maps well to the application domain-specific objects—this is especially true when only part of the document is to be mapped. Using SAX has the additional

benefit of avoiding the creation of an intermediate resource-consuming representation. Finally, SAX is good for implementing stream processing where the business logic is invoked in the midst of document processing. However, SAX can be tedious to use for more complex documents that necessitate managing sophisticated context, and in these cases, developers may find it better to use DOM or JAXB.

In summary, consider using the SAX processing model when any of the following circumstances apply:

- ❒ You are familiar with event-based programming.
- ❒ Your application only consumes documents without making structural modifications to them.
- ❒ The document must only be processed one time.
- ❒ You have to effectively extract and process only parts of the document.
- ❒ Memory usage is an issue or documents may potentially be very large.
- ❒ You want to implement performant stream processing, such as for dealing with very large documents.
- ❒ The structure of a document and the order of its information map well to the domain-specific objects or corresponds to the order in which discrete methods of the application's logic must be invoked. Otherwise, you may have to maintain rather complicated contexts.

Note that the SAX model may not be the best candidate for application developers who are more concerned about implementing business logic.

4.4.1.2 DOM Programming Model

With the DOM programming model, you write code to traverse a tree-like data structure created by the parser from the source document. Typically, processing the XML input data is done in a minimum of two steps, as follows:

1. The DOM parser creates a tree-like data structure that models the XML source document. This structure is called a DOM tree.
2. The application code walks the DOM tree, searching for relevant information that it extracts, consolidates, and processes further. Developers can use consol-

idated information to create domain-specific objects. The cycle of searching for, extracting, and processing the information can be repeated as many times as necessary because the DOM tree persists in memory.

There are limitations to the DOM model. DOM was designed to be both a platform- and language-neutral interface. Because of this, the Java binding of the DOM API is not particularly Java friendly. For example, the binding does not use the `java.util.Collection` API. Generally, DOM is slightly easier to use than the SAX model. However, due to the awkwardness of DOM's Java binding, application developers who are focused on the implementation of the business logic may still find DOM difficult to use effectively. For this reason, similarly with SAX, application developers should be shielded as much as possible from the DOM model.

In addition, the DOM API prior to version level 3 does not support serialization of DOM trees back to XML. Although some implementations do provide serialization features, these features are not standard. Thus, developers should instead rely on XSLT identity transformations, which provide a standard way to achieve serialization back to XML.

Java developers can also use other technologies, such as JDOM and dom4j, which have similar functionality to DOM. The APIs of these technologies tend to be more Java-friendly than DOM, plus they interface well with JAXP. They provide a more elaborate processing model that may alleviate some of DOM's inherent problems, such as its high memory usage and the limitation of processing document content only after a document has been parsed.

Although not yet standard for the Java platform (not until JAXP 1.3), the Xpath API enables using it in conjunction with the DOM programming model. (The Xpath API can be found along with some DOM implementations such as Xerces or dom4j.) Developers use Xpath to locate and extract information from a source document's DOM tree. By allowing developers to specify path patterns to locate element content, attribute values, and subtrees, Xpath not only greatly simplifies, but may even eliminate, tree-traversal code. Since Xpath expressions are strings, they can be easily parameterized and externalized in a configuration file. As a result, developers can create more generic or reusable document processing programs.

To sum up, consider using DOM when any of these circumstances apply:

- ❒ You want to consume or produce documents.
- ❒ You want to manipulate documents and need fine-grained control over the document structure that you want to create or edit.
- ❒ You want to process the document more than once.
- ❒ You want random access to parts of the document. For example, you may want to traverse back and forth within the document.
- ❒ Memory usage is not a big issue.
- ❒ You want to implement data binding but you cannot use JAXB technology because the document either has no schema or it conforms to a DTD schema definition rather than to an XSD schema definition. The document may also be too complex to use SAX to implement data binding. (See "SAX Programming Model" on page 165.)
- ❒ You want to benefit from the flexibility of Xpath and apply Xpath expressions on DOM trees.

4.4.1.3 XML Data-Binding Programming Model

The XML data-binding programming model, contrary to the SAX and DOM models, allows the developer to program the processing of the content of an XML document without being concerned with XML document representations (infosets).

Using a binding compiler, the XML data-binding programming model, as implemented by JAXB, binds components of a source XSD schema to schema-derived Java content classes. JAXB binds an XML namespace to a Java package. XSD schema instance documents can be unmarshalled into a tree of Java objects (called a content tree), which are instances of the Java content classes generated by compiling the schema. Applications can access the content of the source documents using JavaBeans-style get and set accessor methods. In addition, you can create or edit an in-memory content tree, then marshal it to an XML document instance of the source schema. Whether marshalling or unmarshalling, the developer can apply validation to ensure that either the source document or the document about to be generated satisfy the constraints expressed in the source schema.

The steps for using JAXB technology schema-derived classes to process an incoming XML document are very simple, and they are as follows:

1. Set up the JAXB context (`JAXBContext`) with the list of schema-derived packages that are used to unmarshal the documents.
2. Unmarshal an XML document into a content tree. Validation of the document is performed if enabled by the application.
3. You can then directly apply the application's logic to the content tree. Or, you can extract and consolidate information from the content tree and then apply the application's logic on the consolidated information. As described later, this consolidated information may very well be domain-specific objects that may expose a more adequate, schema-independent interface.

This programming model also supports serialization to XML, or marshalling a content tree to an XML format. Marshalling of a document has the following steps:

1. Modify an existing content tree, or create a new tree, from the application's business logic output.
2. Optionally, validate in-memory the content tree against the source schema. Validation is performed in-memory and can be applied independently of the marshalling process.
3. Marshal the content tree into an XML document.

There are various ways a developer can design an application with the schema-derived classes that JAXB generates:

1. The developer may use them directly in the business logic, but, as noted in "Choosing Processing Models" on page 151, this tightly binds the business logic to the schemas from which the classes were generated. This type of usage shares most of the issues of a document-centric processing model.
2. The developer can use the schema-derived classes in conjunction with an object-centric processing model:
   a. The developer may design domain-specific classes whose instances will be populated from the content objects created by unmarshalling an XML document, and vice versa.
   b. The developer may design domain-specific classes, which inherit from the schema-derived classes, and define additional domain-oriented methods. The problem with this design is that these classes are tightly coupled to the

implementation of the schema-derived classes, and they may also expose the methods from the schema-derived classes as part of the domain-specific class. Additionally, as a side effect, if the developer is not careful, this may result in tightly binding the business logic to the schemas from which the classes were generated.

c. The developer can use aggregation or composition and design domain-specific classes that only expose domain-oriented methods and delegate to the schema-derived classes. Since the domain-specific classes only depend on the interfaces of the schema-derived classes, the interfaces of the domain-specific classes may therefore not be as sensitive to changes in the original schema.

Note that when no well-defined schema or variable in nature (abstract) or in number (including new schema revisions) is available, JAXB may be cumbersome to use due to the tight coupling between the schemas and the schema-derived classes. Also note that the more abstract the schema, the less effective the binding.

Consider using the XML data-binding programming model, such as JAXB, when you have any of the following conditions:

- ❒ You want to deal directly with plain Java objects and do not care about, nor want to handle, document representation.
- ❒ You are consuming or producing documents.
- ❒ You do not need to maintain some aspects of a document, such as comments and entity references. The JAXB specification does not require giving access to the underlying document representation (infoset). For example, the JAXB reference implementation is based on SAX 2.0 and therefore does not maintain an underlying document representation. However, other implementations may be layered on top of a DOM representation. A developer may fall back on this DOM representation to access unexposed infoset elements.
- ❒ You want to process the content tree more than once.
- ❒ You want random access to parts of the document. For example, you may want to traverse back and forth within the document.
- ❒ Memory usage may be less of an issue. A JAXB implementation, such as the standard implementation, creates a Java representation of the content of a document that is much more compact than the equivalent DOM tree. The standard

implementation is layered on top of SAX 2.0 and does not maintain an additional underlying representation of the source document. Additionally, where DOM represents all XML schema numeric types as strings, JAXB's standard implementation maps these values directly to much more compact Java numeric data types. Not only does this use less memory to represent the same content, the JAXB approach saves time, because converting between the two representations is not necessary.

- ❒ You previously were implementing XML data-binding manually with DOM, and an XSD schema definition is available.

4.4.1.4 XSLT Programming Model

XSLT is a higher-level processing model than the SAX, DOM, and XML data-binding models. Although developers can mimic XSLT by implementing transformations programmatically on top of SAX, DOM, or the XML data-binding model, XSLT does not compare with other processing models and should be regarded as complementary, to be used along with these other models.

XSLT implements a functional declarative model as opposed to a procedural model. This requires skills that are quite different from Java programming skills. For the most part, XSLT requires developers to code rules, or templates, that are applied when specified patterns are encountered in the source document. The application of the rules adds new fragments or copies fragments from the source tree to a result tree. The patterns are expressed in the Xpath language, which is used to locate and extract information from the source document.

Instead of writing Java code (as with SAX, DOM, and the XML data-binding model), developers using XSLT principally write style sheets, which are themselves XML documents. (Invoking the XSLT engine, however, does require the developer to write Java code.) Compared to the other programming models, XSLT programming gives developers the sort of flexibility that comes with scripting. In an XML-based application, XSLT processing is usually used along with the other three processing models. The XSLT API available with JAXP provides an abstraction for the source and result of transformations, allowing the developer not only the ability to chain transformations but also to interface with other processing models, such as SAX, DOM, and JAXB technology. To interface with SAX and DOM, use the classes `SAXSource`, `SAXResult`, `DOMSource`, and `DOMResult` provided by JAXP. To interface with JAXB, use the classes `JAXBSource` and `JAXBResult`.

By definition, XSLT supports not only processing XML input documents but it also can output XML documents. (Other output methods include text, HTML, and so forth.) Note that although the DOM version level 2 API does not support serialization—that is, transformation of a DOM tree to an XML document—the JAXP implementation of XSLT addresses the serialization of a DOM tree using an identity transformer. An identity transformer copies a source tree to a result tree and applies the specified output method, thus solving the serialization problem in an easy, implementation-independent manner. For example, to output in XML, the output method is set to `xml`. XSLT can also be used to serialize to XML from DOM trees, SAX events, and so forth.

Consider using XSLT when any of the following circumstances apply:

- ❒ You want to change the structure, insert, remove, rename, or filter content of an XML document.
- ❒ You potentially have more than one transformation for the same document. Although one transformation can be hand coded using another API, multiple transformations, because of the scripting nature of style sheets, are better done using XSLT transformations.
- ❒ You have to perform complex transformations. Because of XSLT's functional declarative model, it is easier to design complex transformations by coding individual rules or templates than by hard-coding procedures.
- ❒ You want the ability to be flexible and leave room for future changes in the schemas of documents you are processing.
- ❒ You want to process documents that contain a significant amount of data to minimize performance overhead.
- ❒ You need to transform a document for non-interactive presentation or in batch mode. The performance overhead of transformation is usually less of an issue with non-interactive presentations. Such a document might be a purchase order or an invoice, for example.
- ❒ You must support multiple external schemas but you want to internally program only against a generic schema (schema adapter).

- You want to promote the separation of skills between XML transformation style sheet developers and business logic developers.
- In general, when you must deal with non-interactive presentation or you must integrate various XML data sources or perform XML data exchanges.

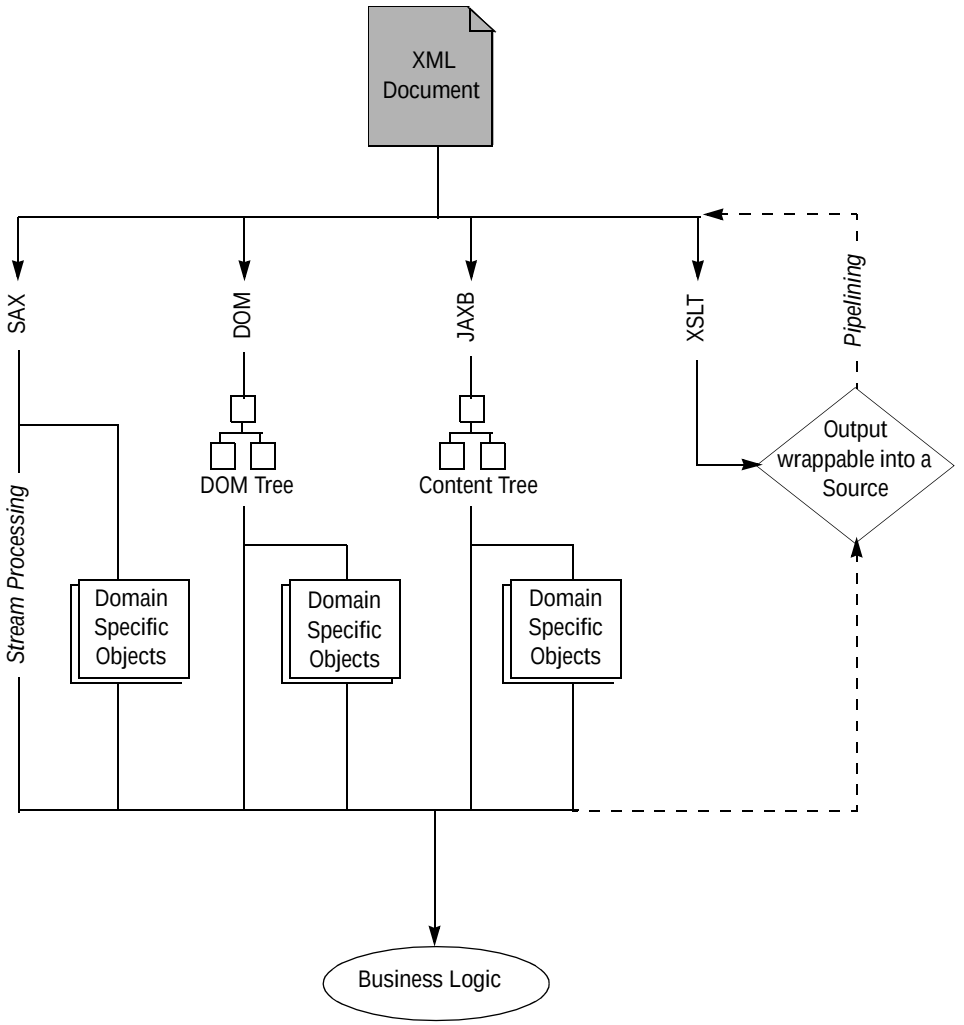

Figure 4.13 Programming Models and Implied Intermediary Representations

4.4.1.5 Recommendation Summary

In summary, choose the programming model and API processing according to your needs. If you need to deal with the content and structure of the document, consider using DOM and SAX because they provide more information about the document itself than JAXB usually does. On the other hand, if your focus is more on the actual, domain-oriented objects that the document represents, consider using JAXB, since JAXB hides the details of unmarshalling, marshalling, and validating the document. Developers should use JAXB—XML data-binding—if the document content has a representation in Java that is directly usable by the application (that is, close to domain-specific objects).

DOM, when used in conjunction with XPath, can be a very flexible and powerful tool when the focus is on the content and structure of the document. DOM may be more flexible than JAXB when dealing with documents whose schemas are not well-defined.

Finally, use XSLT to complement the three other processing models, particularly in a pre- or post-processing stage.

Figure 4.13 summarizes the different programming models from which the developer can choose and highlights the intermediary representations (which have a direct impact on performance) implied by each of them. Table 4.1 summarizes the features of the three most prevalent XML programming models.

Table 4.1 DOM, SAX, and XML Data-Binding Programming Models

DOM	SAX	XML Data-Binding
Tree traversal model	Event-based model	Java-bound content tree model
Random access (in-memory data structure) using generic (application independent) API	Serial access (flow of events) using parameters passed to events	Random access (in-memory data structure) using JavaBeans style accessors

Table 4.1 DOM, SAX, and XML Data-Binding Programming Models (*continued*)

DOM	SAX	XML Data-Binding
High memory usage (The document is often completely loaded in memory, though some techniques such as deferred or lazy DOM node creation may lower the memory usage.)	Low memory usage (only events are generated)	Intermediate memory usage (The document is often completely loaded in memory, but the Java representation of the document is more effective than a DOM representation. Nevertheless, some implementations may implement techniques to lower the memory usage.)
To edit the document (processing the in-memory data structure)	To process parts of the document (handling relevant events)	To edit the document (processing the in-memory data structure)
To process multiple times (document loaded in memory)	To process the document only once (transient flow of events)	To process multiple times (document loaded in memory)
Processing once the parsing is finished	Stream processing (start processing before the parsing is finished, and even before the document is completely read)	Processing once the parsing is finished

4.4.2 Combining XML Processing Techniques

The JAXP API provides support for chaining XML processings: The JAXP `javax.xml.Source` and `javax.xml.Result` interfaces constitute a standard mechanism for chaining XML processings. There are implementations of these two interfaces for DOM, SAX, and even streams; JAXB and other XML processing technologies, such as JDOM and dom4j, provide their own implementations as well.

Basically, XML processings can be chained according to two designs:

- The first design wraps the result of one processing step into a `Source` object that can be processed by the next step. XML processing techniques that can produce an in-memory representation of their results, such as DOM and JAXB, lend themselves to this design. This design is often called “batch sequential” because each processing step is relatively independent and each runs to completion until the next step begins.
- The second design, called “stream processing” or “pipes and filters,” creates a

chain of filters and each filter implements a processing step. XML processing techniques such as SAX work well with this design.

The two designs can, of course, be combined. When combined, transformations and identity transformations become handy techniques to use when the result of a processing step cannot be directly wrapped into a `Source` object compatible with the next step. JAXP also provides support for chaining transformations with the use of `javax.xml.transform.sax.SAXTransformerFactory`.

Code Example 4.10 illustrates an XML processing pipeline that combines SAX and XSLT to validate an incoming purchase order document, extract on-the-fly the purchase order identifier, and transform the incoming document from its external, XSD-based schema to the internal, DTD-based schema supported by the business logic. The code uses a SAX filter chain as the `Source` of a transformation. Alternatively, the code could have used a `SAXTransformerFactory` to create an `org.xml.sax.XMLFilter` to handle the transformation and then chain it to the custom `XMLFilter`, which extracts the purchase order identifier.

```
public class SupplierOrderXDE extends
          XMLDocumentEditor.DefaultXDE {
    public static final String DEFAULT_ENCODING = "UTF-8";
    private XMLFilter filter;
    private Transformer transformer;
    private Source source = null;
    private String orderId = null;

    public SupplierOrderXDE(boolean validating, ...) {
        // Create a [validating] SAX parser
        SAXParser parser = ...;
        filter = new XMLFilterImpl(parser.getXMLReader()) {
            // Implements a SAX XMLFilter that extracts the OrderID
            // element value and assigns it to the orderId attribute
        };
        // Retrieve the style sheet as a stream
        InputStream stream = ...;
        // Create a transformer from the stylesheet
        transformer = TransformerFactory.newInstance()
            .newTransformer(new StreamSource(stream));
    }
    // Sets the document to be processed
```

```
    public void setDocument(Source source) throws ... {
        this.source = source;
    }
    // Builds an XML processing pipeline (chaining a SAX parser and
    // a style sheet transformer) which validates the source document
    // extracts its orderId, transforms it into a different format,
    // and copies the resulting document into the Result object
    public void copyDocument(Result result) throws ... {
        orderId = null;
        InputSource inputSource
            = SAXSource.sourceToInputSource(source);
        SAXSource saxSource = new SAXSource(filter, inputSource);
        transformer.transform(transformer, saxSource, result);
    }
    // Returns the processed document as a Source object
    public Source getDocument() throws ... {
        return new StreamSource(new StringReader(
            getDocumentAsString()));
    }
    // Returns the processed document as a String object
    public String getDocumentAsString() throws ... {
        ByteArrayOutputStream stream = new ByteArrayOutputStream();
        copyDocument(new StreamResult(stream));
        return stream.toString(DEFAULT_ENCODING);
    }
    // Returns the orderId value extracted from the source document
    public String getOrderId() {
        return orderId;
    }
}
```

Code Example 4.10 Combining SAX and XSLT to Perform XML Processing Steps

4.4.3 Entity Resolution

As mentioned earlier in this chapter, XML documents may contain references to other external XML fragments. Parsing replaces the references with the actual content of these external fragments. Similarly, XML schemas may refer to external type definitions, and these definitions must also be accessed for the schema to be

completely interpreted. (This is especially true if you follow the modular design recommendations suggested in "Designing Domain-Specific XML Schemas" on page 131.) In both cases, an XML processor, in the course of processing a document or a schema, needs to find the content of any external entity to which the document or schema refers. This process of mapping external entity references to their actual physical location is called entity resolution. Note that entity resolution recursively applies to external entity references within parsed external entities.

Entity resolution is particularly critical for managing the XML schemas upon which your application is based. As noted in "Validating XML Documents" on page 139, the integrity and the reliability of your application may depend on the validation of incoming documents against specific schemas—typically the very same schemas used to initially design your application. Your application usually cannot afford for these schemas to be modified in any way, whether by malicious modifications or even legitimate revisions. (For revisions, you should at a minimum assess the impact of a revision on your application.)

Therefore, you may want to keep your own copies of the schemas underlying your application and redirect references to these copies. Or you may want to redirect any such references to trusted repositories. A custom entity resolution allows you to implement the desired mapping of external entity references to actual trusted physical locations. Moreover, implementing an entity catalog—even as simple as the one presented in Code Example 4.12—gives you more flexibility for managing the schemas upon which your application depends. Note that to achieve our overall goal, the entity catalog must itself be adequately protected. Additionally, redirecting references to local copies of the schemas may improve performance when compared to referring to remote copies, especially for remote references across the Internet. As described in "Reduce the Cost of Referencing External Entities" on page 189, performance can be improved further by caching in memory the *resolved* entities.

Code Example 4.11 illustrates an entity resolver that implements an interface from the SAX API (`org.xml.sax.EntityResolver`). The entity resolver uses a simple entity catalog to map external entity references to actual locations. The entity catalog is simply implemented as a `Properties` file that can be loaded from a URL (see Code Example 4.12). When invoked, this entity resolver first tries to use the catalog to map the declared public identifier or URI of the external entity to an actual physical location. If this fails—that is, if no mapping is defined in the catalog for this public identifier or URI—the entity resolver uses the declared system identifier or URL of the external entity for its actual physical location. In both cases, the resolver interprets the actual physical location either as an URL or,

as a fall-back, as a Java resource accessible from the class path of the entity resolver class. The latter case allows XML schemas to be bundled along with their dependent XML processing code. Such bundling can be useful when you must absolutely guarantee the consistency between the XML processing code and the schemas.

```
public class CustomEntityResolver implements EntityResolver {
    private Properties entityCatalog = null;

    public CustomEntityResolver(URL entityCatalogURL)
            throws IOException {
        entityCatalog = new Properties(entityCatalog);
        entityCatalog.load(entityCatalogURL.openStream());
    }
    // Opens the physical location as a plain URL or if this fails, as
    // a Java resource accessible from the class path.
    private InputSource openLocation(String location)
            throws IOException {
        URL url = null;
        InputStream entityStream = null;
        try { // Wellformed URL?
            url = new URL(location);
        } catch (MalformedURLException exception) { ... }
        if (url != null) { // Wellformed URL.
            try { // Try to open the URL.
                entityStream = url.openStream();
            } catch (IOException exception) { ... }
        }
        if (entityStream == null) { // Not a URL or not accessible.
            try { // Resource path?
                String resourcePath = url != null
                    ? url.getPath() : location;
                entityStream
                    = getClass().getResourceAsStream(resourcePath);
            } catch (Exception exception1) { ... }
        }
        if (entityStream != null) { // Readable URL or resource.
            InputSource source = new InputSource(entityStream);
            source.setSystemId(location);
```

```
            return source;
        }
        return null;
    }
    // Maps an external entity URI or Public Identifier to a
    // physical location.
    public String mapEntityURI(String entityURI) {
        return entityCatalog.getProperty(entityURI);
    }

    public InputSource resolveEntity(String entityURI,
            String entityURL) {
        InputSource source = null;
        try {
            // Try first to map its URI/PublicId using the catalog.
            if (entityURI != null) {
                String mappedLocation = mapEntityURI(entityURI);
                if (mappedLocation != null) {
                    source = openLocation(mappedLocation);
                    if (source != null) { return source; }
                }
            }
            // Try then to access the entity using its URL/System Id.
            if (entityURL != null) {
                source = openLocation(entityURL);
                if (source != null) { return source; }
            }
        }
    } catch (Exception exception) { ... }
        return null; // Let the default entity resolver handle it.
    }
}
```

Code Example 4.11 Entity Resolver Using a Simple Entity Catalog

```
# DTD Public Indentifier to physical location (URL or resource path)
-//Sun Microsystems, Inc. -
 J2EE Blueprints Group//DTD LineItem 1.0//EN: /com/sun/j2ee/blue-
prints/xmldocuments/rsrc/schemas/LineItem.dtd
```

```
-//Sun Microsystems, Inc. -
 J2EE Blueprints Group//DTD Invoice 1.0//EN: /com/sun/j2ee/blue-
prints/xmldocuments/rsrc/schemas/Invoice.dtd
# XSD URI to physical location (URL or resource path)
http\://blueprints.j2ee.sun.com/LineItem: /com/sun/j2ee/blue-
prints/xmldocuments/rsrc/schemas/LineItem.xsd
http\://blueprints.j2ee.sun.com/Invoice: /com/sun/j2ee/blue-
prints/xmldocuments/rsrc/schemas/Invoice.xsd
```

Code Example 4.12 A Simple Entity Catalog

Although simple implementations such as Code Example 4.11 solve many of the common problems related to external entity management, developers should bear in mind that better solutions are on the horizon. For example, organizations such as the Oasis Consortium are working on formal XML catalog specifications. (See the Web site `http://www.oasis-open.org` for more information on entity resolution and XML catalogs.)

In summary, you may want to consider implementing a custom entity resolution—or, even better, resort to a more elaborate XML catalog solution—in the following circumstances:

- To protect the integrity of your application against malicious modification of external schemas by redirecting references to secured copies (either local copies or those on trusted repositories)
- During design and, even more so, during production, to protect your application against unexpected, legitimate evolution of the schemas upon which it is based. Instead, you want to defer accounting for this evolution until after you have properly assessed their impact on your application.
- To improve performance by maintaining local copies of otherwise remotely accessible schemas.

4.5 Performance Considerations

It is important to consider performance when processing XML documents. XML document processing—handling the document in a pre- or post-processing stage to an application's business logic—may adversely affect application performance

because such processing is potentially very CPU, memory, and input/output or network intensive.

Why does XML document processing potentially impact performance so significantly? Recall that processing an incoming XML document consists of multiple steps, including parsing the document; optionally validating the document against a schema (this implies first parsing the schema); recognizing, extracting, and directly processing element contents and attribute values; or optionally mapping these components to other domain-specific objects for further processing. These steps must occur before an application can apply its business logic to the information retrieved from the XML document. Parsing an XML document often requires a great deal of encoding and decoding of character sets, along with string processing. Depending on the API that is used, recognition and extraction of content may consist of walking a tree data structure, or it may consist of intercepting events generated by the parser and then processing these events according to some context. An application that uses XSLT to preprocess an XML document adds more processing overhead before the real business logic work can take place. When the DOM API is used, it creates a representation of the document—a DOM tree—in memory. Large documents result in large DOM trees and corresponding consumption of large amounts of memory. The XML data-binding process has, to some extent, the same memory consumption drawback. Many of these constraints hold true when generating XML documents.

There are other factors with XML document processing that affect performance. Often, the physical and logical structures of an XML document may be different. An XML document may also contain references to external entities. These references are resolved and substituted into the document content during parsing, but prior to validation. Given that the document may originate on a system different from the application's system, and external entities—and even the schema itself—may be located on remote systems, there may be network overhead affecting performance. To perform the parsing and validation, external entities must first be loaded or downloaded to the processing system. This may be a network intensive operation, or require a great deal of input and output operations, when documents have a complex physical structure.

In summary, XML processing is potentially CPU, memory, and network intensive, for these reasons:

- It may be CPU intensive. Incoming XML documents need not only to be parsed but also validated, and they may have to be processed using APIs which may themselves be CPU intensive. It is important to limit the cost of validation

as much as possible without jeopardizing the application processing and to use the most appropriate API to process the document.

- It may be memory intensive. XML processing may require creating large numbers of objects, especially when dealing with document object models.
- It may be network intensive. A document may be the aggregation of different external entities that during parsing may need to be retrieved across the network. It is important to reduce as much as possible the cost of referencing external entities.

Following are some guidelines for improving performance when processing XML documents. In particular, these guidelines examine ways of improving the CPU, memory, and input/output or network consumption.

4.5.1 Limit Parsing of Incoming XML Documents

In general, it is best to parse incoming XML documents only when the request has been properly formulated. In the case of a Web service application, if a document is retrieved as a `Source` parameter from a request to an endpoint method, it is best first to enforce security and validate the meta information that may have been passed as additional parameters with the request.

In a more generic messaging scenario, when a document is wrapped inside another document (considered an envelope), and the envelope contains meta information about security and how to process the inner document, you may apply the same recommendation: Extract the meta information from the envelope, then enforce security and validate the meta information before proceeding with the parsing of the inner document. When implementing a SAX handler and assuming that the meta information is located at the beginning of the document, if either the security or the validation of the meta information fails, then the handler can throw a SAX exception to immediately abort the processing and minimize the overall impact on performance.

4.5.2 Use the Most Appropriate API

It's important to choose the most appropriate XML processing API for your particular task. In this section, we look at the different processing models in terms of the situations in which they perform best and where their performance is limited.

In general, without considering memory consumption, processing using the DOM API tends to be slower than processing using the SAX API. This is because

DOM may have to load the entire document into memory so that the document can be edited or data retrieved, whereas SAX allows the document to be processed as it is parsed. However, despite its initial slowness, it is better to use the DOM model when the source document must be edited or processed multiple times.

You should also try to use JAXB whenever the document content has a direct representation, as domain-specific objects, in Java. If you don't use JAXB, then you must manually map document content to domain-specific objects, and this process often (when SAX is too cumbersome to apply—see page 166) requires an intermediate DOM representation of the document. Not only is this intermediate DOM representation transient, it consumes memory resources and must be traversed when mapping to the domain-specific objects. With JAXB, you can automatically generate the same code, thus saving development time, and, depending on the JAXB implementation, it may not create an intermediate DOM representation of the source document. In any case, JAXB uses less memory resources as a JAXB content tree is by nature smaller than an equivalent DOM tree.

When using higher-level technologies such as XSLT, keep in mind that they may rely on lower-level technologies like SAX and DOM, which may affect performance, possibly adversely.

When building complex XML transformation pipelines, use the JAXP class `SAXTransformerFactory` to process the results of one style sheet transformation with another style sheet. You can optimize performance—by avoiding the creation of in-memory data structures such as DOM trees—by working with SAX events until at the last stage in the pipeline.

As an alternative, you may consider using APIs other than the four discussed previously. JDOM and dom4j are particularly appropriate for applications that implement a document-centric processing model and that must manipulate a DOM representation of the documents.

JDOM, for example, achieves the same results as DOM but, because it is more generic, it can address any document model. Not only is it optimized for Java, but developers find JDOM easy to use because it relies on the Java `Collection` API. JDOM documents can be built directly from, and converted to, SAX events and DOM trees, allowing JDOM to be seamlessly integrated in XML processing pipelines and in particular as the source or result of XSLT transformations.

Another alternative API is dom4j, which is similar to JDOM. In addition to supporting tree-style processing, the dom4j API has built-in support for Xpath. For example, the `org.dom4j.Node` interface defines methods to select nodes according to an Xpath expression. dom4j also implements an event-based pro-

cessing model so that it can efficiently process large XML documents. When Xpath expressions are matched during parsing, registered handlers can be called back, thus allowing you to immediately process and dispose of parts of the document without waiting for the entire document to be parsed and loaded into memory.

When receiving documents through a service endpoint (either a JAX-RPC or EJB service endpoint) documents are parsed as abstract `Source` objects. As already noted, do not assume a specific implementation—`StreamSource`, `SAXSource`, or `DOMSource`—for an incoming document. Instead, you should ensure that the optimal API is used to bridge between the specific `Source` implementation passed to the endpoint and the intended processing model. Keep in mind that the JAXP XSLT API does not guarantee that identity transformations are applied in the most effective way. For example, when applying an identity transformation from a DOM tree to a DOM tree, the most effective way is to return the source tree as the result tree without further processing; however, this behavior is not enforced by the JAXP specification.

A developer may also want to implement stream processing for the application so that it can receive the processing requirements as part of the SOAP request and start processing the document before it is completely received. Document processing in this manner improves overall performance and is useful when passing very large documents. Extreme caution should be taken if doing this, since there is no guarantee that the underlying JAX-RPC implementation will not wait to receive the complete document before passing the `Source` object to the endpoint and that it will effectively pass a `Source` object that allows for stream processing, such as `StreamSource` or `SAXSource`. The same holds true when implementing stream processing for outgoing documents. While you can pass a `Source` object that allows for stream processing, there is no guarantee on how the underlying JAX-RPC implementation will actually handle it.

4.5.3 Choose Effective Parser and Style Sheet Implementations

Each parser and style sheet engine implementation is different. For example, one might emphasize functionality, while another performance. A developer might want to use different implementations depending on the task to be accomplished. Consider using JAXP, which not only supports many parsers and style sheet engines, but also has a pluggability feature that allows a developer to swap between implementations and select the most effective implementation for an application's requirements.

When you use JAXP, you can later change the underlying parser implementation without having to change application code.

4.5.3.1 Tune Underlying Parser and Style Sheet Engine Implementations

The JAXP API defines methods to set and get features and properties for configuring the underlying parser and style sheet engine implementations. A particular parser, document builder, or transformer implementation may define specific features and properties to switch on or off specific behaviors dedicated to performance improvement. These are separate from such standard properties and features, such as the `http://xml.org/sax/features/validation` feature used to turn validation on or off.

For example, Xerces defines a deferred expansion feature called `http://apache.org/xml/features/dom/defer-node-expansion`, which enables or disables a lazy DOM mode. In lazy mode (enabled by default), the DOM tree nodes are lazily evaluated, their creation is deferred: They are created only when they are accessed. As a result, DOM tree construction from an XML document returns faster since only accessed nodes are expanded. This feature is particularly useful when processing only parts of the DOM tree. Grammar caching, another feature available in Xerces, improves performance by avoiding repeated parsing of the same XML schemas. This is especially useful when an application processes a limited number of schemas, which is typically the case with Web services.

Use care when setting specific features and properties to preserve the interchangeability of the underlying implementation. When the underlying implementation encounters a feature or a property that it does not support or recognize, the `SAXParserFactory`, the `XMLReader`, or the `DocumentBuilderFactory` may throw these exceptions: a `SAXNotRecognizedException`, a `SAXNotSupportedException`, or an `IllegalArgumentException`. Avoid grouping unrelated features and properties, especially standard versus specific ones, in a single `try/catch` block. Instead, handle exceptions independently so that optional specific features or properties do not prevent switching to a different implementation. You may design your application in such a way that features and properties specific to the underlying implementations may also be defined externally to the application, such as in a configuration file.

4.5.3.2 Reuse and Pool Parsers and Style Sheets

An XML application may have to process different types of documents, such as documents conforming to different schemas. A single parser may be used (per thread of execution) to handle successively documents of different types just by reassigning the handlers according to the source documents to be processed. Parsers, which are complex objects, may be pooled so that they can be reused by other threads of execution, reducing the burden on memory allocation and garbage collection. Additionally, if the number of different document types is large and if the handlers are expensive to create, handlers may be pooled as well. The same considerations apply to style sheets and transformers.

Parsers, document builders, and transformers, as well as style sheets, can be pooled using a custom pooling mechanism. Or, if the processing occurs in the EJB tier, you may leverage the EJB container's instance pooling mechanism by implementing stateless session beans or message-driven beans dedicated to these tasks. Since these beans are pooled by the EJB container, the parsers, document builders, transformers, and style sheets to which they hold a reference are pooled as well.

Style sheets can be compiled into `javax.xml.transform.Templates` objects to avoid repeated parsing of the same style sheets. `Templates` objects are thread safe and are therefore easily reusable.

4.5.4 Reduce Validation Cost

Not only is it important, but validation may be required to guarantee the reliability of an XML application. An application may legitimately rely on the parser's validation so that it can avoid double-checking the validity of document contents. Validation is an important step of XML processing, but keep in mind that it may affect performance.

Consider the trusted and reliable system depicted in Figure 4.14. This system is composed of two loosely coupled applications. The front-end application receives XML documents as part of requests and forwards these documents to the reservation engine application, which is implemented as a document-centric workflow.

Although you must validate external incoming XML documents, you can exchange freely—that is, without validation—internal XML documents or already validated external XML documents. In short, you need to validate only at the system boundaries, and you may use validation internally only as an assertion mechanism during development. You may turn validation off when in production and looking for optimal performance.

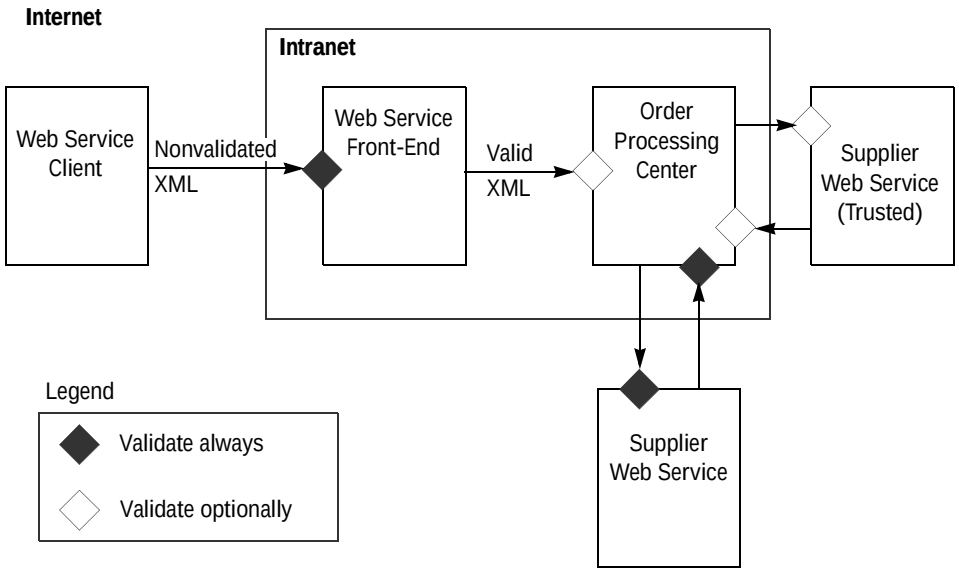

Figure 4.14 Validating Only When Necessary

In other words, when you are both the producer and consumer of XML documents, you may use validation as an assertion mechanism during development, then turn off validation when in production. Additionally, during production validation can be used as a diagnostic mechanism by setting up validation so that it is triggered by fault occurrences.

4.5.5 Reduce the Cost of Referencing External Entities

Recall that an XML document may be the aggregation of assorted external entities, and that these entities may need to be retrieved across the network when parsing. In addition, the schema may also have to be retrieved from an external location. External entities, including schemas, must be loaded and parsed even when they are not being validated to ensure that the same information is delivered to the application regardless of any subsequent validation. This is especially true with respect to default values that may be specified in an incoming document schema.

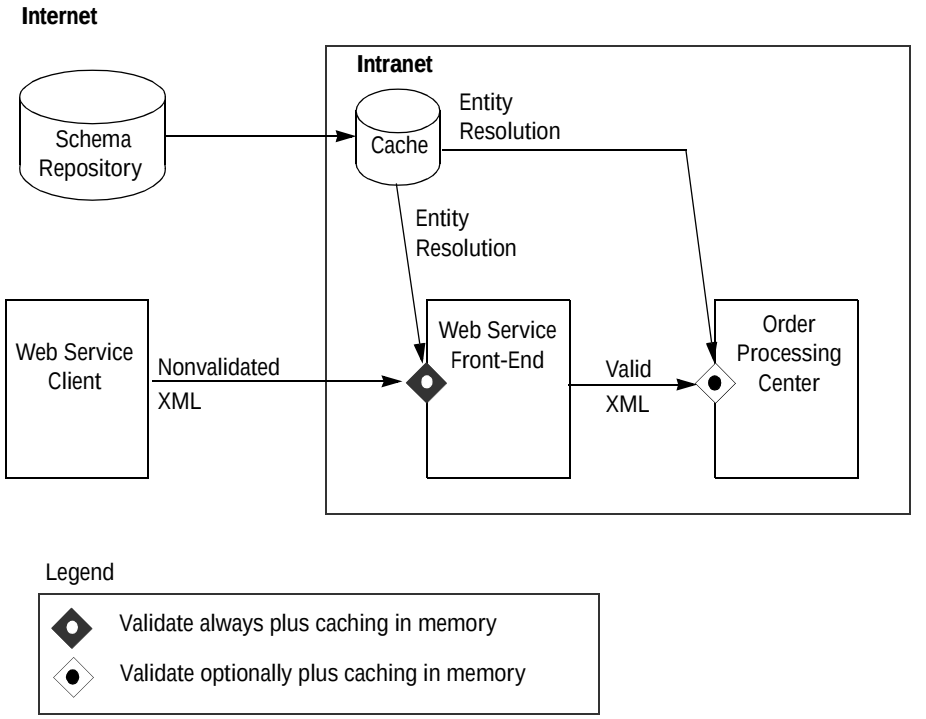

Figure 4.15 An Architecture for Caching External Entities

There are two complementary ways to reduce the cost of referencing external entities:

1. **Caching using a proxy cache**—You can improve the efficiency of locating references to external entities that are on a remote repository by setting up a proxy that caches retrieved, external entities. However, references to external entities must be URLs whose protocols the proxy can handle. (See Figure 4.15, which should be viewed in the context of Figure 4.14.)
2. **Caching using a custom entity resolver**—SAX parsers allow XML applications to handle external entities in a customized way. Such applications have to register their own implementation of the `org.xml.sax.EntityResolver` interface with the parser using the `setEntityResolver` method. The applications are then able to intercept external entities (including schemas) before they are parsed. Similarly, JAXP defines the `javax.xml.transform.URIResolver` inter-

face. Implementing this interface enables you to retrieve the resources referred to in the style sheets by the `xsl:import` or `xsl:include` statements. For an application using a large set of componentized style sheets, this may be used to implement a cache in much the same way as the `EntityResolver`. You can use `EntityResolver` and `URIResolver` to implement:

- A caching mechanism in the application itself, or
- A custom URI lookup mechanism that may redirect system and public references to a local copy of a public repository.

You can use both caching approaches together to ensure even better performance. Use a proxy cache for static entities whose lifetime is greater than the application's lifetime. This particularly works with public schemas, which include the version number in their public or system identifier, since they evolve through successive versions. A custom entity resolver may first map public identifiers (usually in the form of a URI) into system identifiers (usually in the form of an URL). Afterwards, it applies the same techniques as a regular cache proxy when dealing with system identifiers in the form of an URL, especially checking for updates and avoiding caching dynamic content. Using these caching approaches often results in a significant performance improvement, especially when external entities are located on the network. Code Example 4.13 illustrates how to implement a caching entity resolver using the SAX API.

```
import java.util.Map;
import java.util.WeakHashMap;
import java.lang.ref.SoftReference;
import org.xml.sax.*;

public class CachingEntityResolver implements EntityResolver {
    public static final int MAX_CACHE_ENTRY_SIZE = ...;
    private EntityResolver parentResolver;
    private Map entities = new WeakHashMap();

    static private class Entity {
        String name;
        byte[] content;
    }

    public CachingEntityResolver(EntityResolver parentResolver) {
        this.parentResolver = parentResolver;
```

```
        buffer = new byte[MAX_CACHE_ENTRY_SIZE];
    }

    public InputSource resolveEntity(String publicId,
        String systemId) throws IOException, SAXException {
        InputStream stream = getEntity(publicId, systemId);
        if (stream != null) {
            InputSource source = new InputSource(stream);
            source.setPublicId(publicId);
            source.setSystemId(systemId);
            return source;
        }
        return null;
    }

    private InputStream getEntity(String publicId, String systemId)
            throws IOException, SAXException {
        Entity entity = null;
        SoftReference reference
            = (SoftReference) entities.get(systemId);
        if (reference != null) {
            // Got a soft reference to the entity,
            // let's get the actual entity.
            entity = (Entity) reference.get();
        }
        if (entity == null) {
            // The entity has been reclaimed by the GC or was
            // never created, let's download it again! Delegate to
            // the parent resolver that implements the actual
            // resolution strategy.
            InputSource source
                = parentResolver.resolveEntity(publicId, systemId);
            if (source != null) {
                return cacheEntity(publicId, systemId,
                    source.getByteStream());
            }
            return null;
        }
        return new ByteArrayInputStream(entity.content);
```

```
        }
        // Attempts to cache an entity; if it's too big just
        // return an input stream to it.
        private InputStream cacheEntity(String publicId,
                String systemId, InputStream stream) throws IOException {
            stream = new BufferedInputStream(stream);
            int count = 0;
            for (int i = 0; count < buffer.length; count += i) {
                if ((i = stream.read(buffer, count,
                    buffer.length - count)) < 0) { break; }
            }
            byte[] content = new byte[count];
            System.arraycopy(buffer, 0, content, 0, count);
            if (count != buffer.length) {
                // Cache the entity for future use, using a soft reference
                // so that the GC may reclaim it if it's not referenced
                // anymore and memory is running low.
                Entity entity = new Entity();
                entity.name = publicId != null ? publicId : systemId;
                entity.content = content;
                entities.put(entity.name, new SoftReference(entity));
                return new ByteArrayInputStream(content);
            }
            // Entity too big to be cached.
            return new SequenceInputStream(
                new ByteArrayInputStream(content), stream);
        }
    }
```

Code Example 4.13 Using SAX API to Implement a Caching Entity Resolver

4.5.6 Cache Dynamically Generated Documents

Dynamically generated documents are typically assembled from values returned from calls to business logic. Generally, it is a good idea to cache dynamically generated XML documents to avoid having to refetch the document contents, which entails extra round trips to a business tier. This is a good rule to follow when the data is predominantly read only, such as catalog data. Furthermore, if applicable, you can

cache document content (DOM tree or JAXB content tree) in the user's session on the interaction or presentation layer to avoid repeatedly invoking the business logic.

However, you quickly consume more memory when you cache the result of a user request to serve subsequent, related requests. When you take this approach, keep in mind that it must not be done to the detriment of other users. That is, be sure that the application does not fail because of a memory shortage caused by holding the cached results. To help with memory management, use soft references, which allow more enhanced interaction with the garbage collector to implement caches.

When caching a DOM tree in the context of a distributed Web container, the reference to the tree stored in an HTTP session may have to be declared as `transient`. This is because `HttpSession` requires objects that it stores to be Java serializable, and not all DOM implementations are Java serializable. Also, Java serialization of a DOM tree may be very expensive, thus countering the benefits of caching.

4.5.7 Use XML Judiciously

Using XML documents in a Web services environment has its pluses and minuses. XML documents can enhance Web service interoperability: Heterogeneous, loosely coupled systems can easily exchange XML documents because they are text documents. However, loosely coupled systems must pay the price for this ease of interoperability, since the parsing that these XML documents require is very expensive. This applies to systems that are loosely coupled in a technical and an enterprise sense.

Contrast this with tightly coupled systems. System components that are tightly coupled can use standard, nondocument-oriented techniques (such as RMI) that are far more efficient in terms of performance and require far less coding complexity. Fortunately, with technologies such as JAX-RPC and JAXB you can combine the best of both worlds. Systems can be developed that are internally tightly coupled and object oriented, and that can interact in a loosely coupled, document-oriented manner.

Generally, when using XML documents, follow these suggestions:

- ❒ Rely on XML protocols, such as those implemented by JAX-RPC and others, to interoperate with heterogeneous systems and to provide loosely coupled integration points.
- ❒ Avoid using XML for unexposed interfaces or for exchanges between components that should be otherwise tightly coupled.

Direct Java serialization of domain-specific objects is usually faster than XML serialization of an equivalent DOM tree or even the Java serialization of the DOM tree itself (when the DOM implementation supports such a Java serialization). Also, direct Java serialization of the domain-specific objects usually results in a serialized object form that is smaller than the serialized forms of the other two approaches. The Java serialization of the DOM tree is usually the most expensive in processing time as well as in memory footprint; therefore it should be used with extreme care (if ever), especially in an EJB context where serialization occurs when accessing remote enterprise beans. When accessing local enterprise beans, you can pass DOM tree or DOM tree fragments without incurring this processing expense. Table 4.2 summarizes the guidelines for using XML for component interactions.

To summarize, when implementing an application which spans multiple containers, keep the following points in mind (also see Table 4.2).

- ❒ For remote component interaction, Java objects are efficient, serialized XML —although expensive—may be used for interoperability. DOM is very expensive, and Java serialization of DOM trees is not always supported.
- ❒ For local component interaction, Java objects are the most efficient and DOM may be used when required by the application processing model. However, serialized XML is to be avoided.
- ❒ Bind to domain-specific Java objects as soon as possible and process these objects rather than XML documents.

Table 4.2 Guidelines for Using XML Judiciously for Component Interaction

Data Passing Between Components	Remote Components	Local Components
Java Objects	Efficient	Highly efficient (Fine-grained access}
Document Object Model	Very expensive, nonstandard serialization	Only for document-centric architectures
Serialized XML	Expensive, but interoperable	No reason to serialize XML for local calls

4.6 Conclusion

This chapter covered much of the considerations a developer should keep in mind when writing applications, particularly Web services-based applications, that use XML documents. Once these issues and considerations are understood, then a developer can make informed decisions about not only how to design and implement these XML-based applications, but if the applications should even be based on XML. In a sense, the developer is encouraged to ask this question first: "Should this application be based on XML?"

Once the developer decides that XML is appropriate for the application, then the next task is for the developer to design a sound and balanced architecture. Such an architecture should rely on XML only for what XML is good at. That is, the architecture should use XML for open, inter-application communication, configuration descriptions, information sharing, and especially for accessing domains for which public XML schemas exist. At the same time, XML may not be the appropriate solution of choice when application interfaces are not exposed publicly or for exchanging information between components that should be communicating in a tightly coupled manner.

The chapter also considered the factors a developer must weigh when deciding where and how to perform XML processing. Should XML processing be limited to the interaction layer of a Web service as a pre- or post-processing stage of the business logic, or should the business logic itself be expressed in terms of

XML processing? The chapter further helped clarify when it makes sense to implement an object-centric processing model or a document-centric processing model.

These decisions determine the developer's choice among the different XML-processing APIs and implementations. Furthermore, the chapter described the performance trade-offs among these different processing techniques and compared performance with functionality and ease of use. It indicated situations where and why one technique might be superior to another and helped delineate the rationale for these choices.