

Buy the book:

From the back cover

J2EE and XML are important technologies in their own right, but applications that use them together will benefit from their synergy. Java and J2EE make a powerful platform for building robust application logic. XML facilitates flexible data storage and manipulation. Properly using XML with J2EE, you can develop the most powerful enterprise systems that can be built today. This book shows you how.

J2EE and XML Development is a rich yet concise guide. It teaches you how, where, and why to use XML in each layer of your J2EE application. It categorizes and explains many recent Java and XML technologies and ways in which a J2EE application can best use them. It untangles the web of Java APIs for XML, including the JAX family, as well as other popular emerging standards like JDOM, explaining each in terms of its functionality, and illustrating their intended use through examples.

What's inside

- Use XML in component interfaces,
- Build XSLT and XSP presentation layers
- Web Services using SOAP, WSDL, and UDDI
- Learn to use Java APIs for XML, including
 - JAXP,
 - JAXB,
 - JAXM
- Understand XML technologies like
 - XQuery,
 - PDOM,
 - XQL

Kurt Gabrick is an architect and developer with J2EE and XML experience from development projects for Visa USA, IBM, Cisco Systems, Siemens Communications and others. Dave Weiss is a distributed systems developer and trainer of software developers.

To buy this book

Praise for this book

“This book is your survival guide to understand the repertoire of XML based technologies and adopt them in your existing enterprise system.”

– JavaRanch.com

“I am impressed by the broad treatment of related subjects in this book. Not only does it list many aspects of application development, it also covers the important topics of testing and problem tracking ... does an excellent job in itemizing the key points of the whole subject.”

– Compunotes.com

“I recommend this book especially to architects and advanced developers who are interested in adapting their J2EE skills to the new world of XML based web services. It is also a great book for beginners who want to see the big pictures and understand why J2EE/XML is a platform that is worth investing their time and effort.”

– Austin JUG

“!!! Very Good”

– Today's Books

[To buy this book](#)

J2EE and XML Development

KURT A. GABRICK
DAVID B. WEISS



MANNING
Greenwich
(74° w. long.)

To buy this book

For electronic information and ordering of this and other Manning books, go to www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:


Special Sales Department	1-800-247-6553 within the U.S.
Manning Publications Co.	1-419-281-1802 outside the U.S.
209 Bruce Park Avenue	Fax: 1-419-281-6883
Greenwich, CT 06830	email: orders@manning.com

©2002 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books they publish printed on acid-free paper, and we exert our best efforts to that end.

 Manning Publications Co.	Copyeditor: Maarten Reilingh
209 Bruce Park Avenue	Typesetter: Dottie Marsico
Greenwich, CT 06830	Cover designer: Leslie Haimes

ISBN 1-930110-30-8

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 - VHG - 05 04 03 02

To buy this book

*To Maggie—
For your love, patience, and poor taste in men
KAG*

*To My Family—
You have given me an unlimited amount of support and strength.
Thank you for everything.
DBW*

To buy this book

To buy this book

contents

preface xi
acknowledgments xii
about this book xiii
about the authors xvii
about the cover illustration xix
author online xxi

1 *Getting started 1*

- 1.1 Distributed systems overview 2
 - Distributed systems concepts 3* ▪ *N-tier application architecture 12* ▪ *Overcoming common challenges 14*
- 1.2 The J2EE development process 22
 - J2EE and development methodologies 22*
 - J2EE development tools 24*
- 1.3 Testing and deployment in J2EE 29
 - Testing J2EE applications 29*
 - Deploying J2EE applications 33*
- 1.4 Summary 35

- 2 XML and Java 37**
- 2.1 XML and its uses 38
 - XML validation technologies 41* ▪ *XML parsing technologies 44* ▪ *XML translation technologies 46*
 - Messaging technologies 48* ▪ *Data manipulation and retrieval technologies 51* ▪ *Data storage technologies 54*
 - 2.2 The Java APIs for XML 55
 - JAXP 57* ▪ *JDOM 66* ▪ *JAXB 69*
 - Long Term JavaBeans Persistence 74*
 - JAXM 76* ▪ *JAX-RPC 77* ▪ *JAXR 78*
 - 2.3 Summary 78
- 3 Application development 81**
- 3.1 XML component interfaces 82
 - Using value objects 84* ▪ *Implementing XML value objects 87* ▪ *When not to use XML interfaces 95*
 - 3.2 XML and persistent data 96
 - Querying XML data 97* ▪ *Storing XML data 103*
 - When not to use XML persistence 110*
 - 3.3 Summary 110
- 4 Application integration 113**
- 4.1 Integrating J2EE applications 114
 - Traditional approaches to systems integration 114*
 - XML-based systems integration 122*
 - 4.2 A web services scenario 125
 - 4.3 J2EE and SOAP 125
 - Creating a simple SOAP message 126*
 - Using SOAP with Attachments 129*
 - Using JAXM for SOAP Messaging 131*

- 4.4 Building web services in J2EE 138
 - What is a web service?* 139
 - *Providing web services in J2EE* 140
 - *Implementing our example web services* 142
 - *Consuming web services in J2EE* 153
 - J2EE web services and Microsoft .NET* 153
- 4.5 Summary 154

5 **User interface development** 157

- 5.1 Creating a thin-client user interface 158
 - Serving different types of devices* 159
 - *Serving multiple locales* 159
 - *An example to work through* 160
- 5.2 The pure J2EE approach 162
 - The J2EE presentation tool kit* 163
 - Issues in J2EE MVC architecture* 164
 - Building our example in J2EE* 166
 - Analyzing the results* 177
- 5.3 The J2EE/XML approach 177
 - Adding XSLT to the web process flow* 177
 - Analyzing the results* 185
 - Extending to binary formats* 186
- 5.4 XML web publishing frameworks 195
 - Introduction to Cocoon architecture* 196
 - Using Cocoon to render the watch list page* 197
 - Analyzing the results* 200
- 5.5 A word about client-side XSLT 201
- 5.6 Summary 201

6 **Case study** 203

- 6.1 Case study requirements 204
- 6.2 The application environment 206

- 6.3 The analysis phase 207
 - Services and data layer analysis 207* ▪ *Data storage analysis 208* ▪ *Other necessary components 208*
 - 6.4 The design phase 210
 - Designing the application logic layer 210*
 - Designing the user interface 212*
 - 6.5 Validating our design 213
 - 6.6 The implementation phase 215
 - Building the controller servlet 215* ▪ *Building the ApplicationMenu component 217* ▪ *Building the ComponentLocator 218* ▪ *Building the BugAccessorBean 221* ▪ *Building the XSLTFilter 223*
 - 6.7 Structuring application data 224
 - 6.8 The Amaya web service 225
 - 6.9 Running the application 229
 - Installation 229* ▪ *Viewing the main menu 230*
 - Viewing common system problems 231* ▪ *Viewing and updating the Amaya problem list 231* ▪ *Inspecting the web services SOAP messages 232*
 - 6.10 Summary 233
- appendix A Design patterns for J2EE and XML 235*
appendix B Distributed application security 243
appendix C The Ant build tool 249
resources 265
index 269

preface

Enterprise Java development and XML are two of the hottest topics in technology today. Both riddled with acronyms and buzzwords, they are also two of the most poorly understood and abused technologies around. The potential to build platform-neutral, vendor-independent systems has created a flurry of development and a host of new standards. It seems the list of APIs and specifications grows longer and more complex every day.

In early 2000, we decided the time was right to write a book about using XML technology in enterprise Java applications. It occurred to us that many books had been written on either XML or J2EE, but none of them really addressed the subjects together. We also recognized a failing in the content of existing books, which focus heavily on API details and “Hello, world!” examples while skirting the more complex issues of architecture, design tradeoffs, and effective techniques for developing distributed systems.

This book is intended to fill the gap between books on J2EE and those on XML. It demystifies the buzzwords, contains frank discussions on the capabilities and appropriate use of various enterprise Java and XML tools, and provides a logical context for deciding how to structure your XML-enabled J2EE applications. We hope you enjoy it.

acknowledgments

There are a number of people without whom this book would not be possible. We specifically acknowledge:

Our clients past and present, for entrusting their enterprise development efforts to our care and affording us the opportunity to road test the technologies and techniques discussed in this book. There is no substitute for experience in software development, and we thank you for the opportunity.

The developers of the technologies and standards covered in this book, for creating a wealth of patterns and tools to make distributed application development and integration easier for all of us. We especially acknowledge those developers who dedicate their time and energy to open source development efforts that benefit us all.

Our publisher, Marjan Bace, for giving us the opportunity to write a unique book on a complex subject, and our editors and reviewers, for their guidance and encouragement along the way. The editorial and production staff at Manning included Ted Kennedy, Alex Garrett, Maarten Reilingh, Syd Brown, Dottie Marisco, and Mary Piergies. Our reviewers included Randy Akl, Russell Gold, Owen Green, Berndt Hamboeck, Carson Hager, Lee Harding, Allen Hogan, Evan Ireland, Andrew Stevens, David Tillotson, and Jason Weiss. Special thanks to Scott Johnston who reviewed the book for technical accuracy shortly before it went to press.

Our friends and family, for lending all types of support to this effort. We especially thank Maggie Gabrick, who spent many hours translating between code jockey and English during this process.

about this book

This book is about building better applications with Java 2, Enterprise Edition (J2EE) and XML technology. It teaches you how, where, and when to use XML in your J2EE system. It categorizes and explains many recent Java and XML technology developments and suggests ways in which a J2EE application can utilize them.

J2EE and XML are each substantial technologies in their own right. Applications that use them together can realize the benefits of both. J2EE enables the creation of robust and flexible application logic. XML enables powerful data storage, manipulation, and messaging. A J2EE application that makes proper use of XML is one of the most robust component-based systems that you can build.

Beyond identifying areas where XML can play a role in a J2EE application, this book also discusses important tradeoffs to be considered when choosing to build a J2EE application with XML over pure J2EE. The potential drawbacks of using each proposed XML technology are compared with its benefits, allowing you to make an informed decision about its use.

You probably already own a book or two on the topics of J2EE and XML. There are numerous books available to teach you the low level intricacies of J2EE development. There are at least as many on XML and related technologies. There are even a few on the subject of using Java and XML together. Why then should you read this book?

This book will add to what you know, not restate it. It is not a fifteen-hundred-page tome on J2EE with the APIs listed at the back. It is not a detailed

reference on XML either. It is a targeted guide that builds on your existing knowledge of J2EE application development and shows you how to enhance your applications with XML. It will help you build distributed systems that are more robust, manageable, and secure.

The ultimate goal of this book is to arm you with relevant knowledge about the state of J2EE and XML technology and the ways in which they are best put to use. By the end of the book, you should have an excellent idea about which XML technologies you want to use, how you plan to use them, and where to go to learn more about them.

Who should read this book

This is an intermediate-level book and is not a primer on Java, XML, or J2EE. Its primary audience is the distributed application developer. It assumes that you have some practical experience with J2EE and an understanding of XML at the conceptual level. Some basic concepts are briefly introduced as context for detailed discussions, but this book should by no means be your first exposure to either J2EE development or XML. The focus of this book is on the identification, classification, and practical use of important XML-related Java technologies. Getting the most out of this book therefore requires some prior knowledge of J2EE and XML basics.

If you are an application development professional looking for proven approaches to solving complicated problems with J2EE and XML technology, this book is for you. It is a guide to help you make better decisions when designing and building your applications. It presents technical alternatives, provides examples of their implementation, and explains the tradeoffs between them. Discussions are limited to the most relevant topics in each area to maximize the benefits of reading the book and managing its overall length.

How this book is organized

We begin by identifying the common challenges in distributed application development and the design strategies used to overcome them. We discuss how J2EE and the other emerging Java APIs for XML can be implemented to achieve those design goals. We examine the J2EE and XML development process, suggesting some tools and techniques you can employ to build applications most efficiently.

Chapters are dedicated to each layer of an n-tier distributed application, providing in depth coverage of the most recent J2EE/XML developments and usage examples. Additionally, the final chapter presents a detailed case study to synthesize various topics discussed in the book in the context of an end-to-end

To buy this book

J2EE/XML application. The case study illustrates the general approach to J2EE/XML development problems, identifies critical analysis and design decisions, and discusses the benefits and drawbacks associated with those decisions.

Chapter 1: Getting started

This first chapter introduces important concepts, tools, and techniques for building J2EE and XML applications. As a distributed application developer, you face a broad range of challenges as you begin each new project. These challenges range from architectural and design issues to tool selection and development process management.

To overcome these challenges, you require both an appreciation for distributed systems development issues and knowledge of specific tools you can use in a J2EE environment. This chapter summarizes the common challenges to be overcome at each stage of a J2EE and XML project and describes the tools and techniques you need to be successful.

Chapter 2: The Java APIs for XML

In recent months, there has been a flurry of Java community development activity in the area of XML. The result has been the creation of a complex set of closely related XML APIs, each of which is either in specification or development. These APIs include the JAX family, as well as other popular emerging standards like JDOM.

This chapter untangles the web of Java APIs for XML, identifying and classifying each in terms of its functionality, intended use, and maturity. Where possible, we provide usage examples for each new API and describe how it might be best used in your J2EE system. We also identify areas in which the APIs overlap and suggest which ones are likely to be combined or eliminated in the future. Subsequent chapters build upon your understanding of these APIs by providing more specific examples of their implementation.

Chapter 3: Application development

Making changes to J2EE application logic and data structures can be costly and time-consuming. Initial development of a flexible and robust application logic layer is therefore critical to the longevity of your system. This chapter demonstrates how XML technology can help you achieve that goal.

Using XML in component interfaces is covered, as is the use of XML for data storage and retrieval. Examples using common J2EE design patterns such as Value Object and Data Access Object with the Java APIs for XML are provided.

Technologies discussed include JAXB, JDOM, XQuery, PDOM, and XQL. Design tradeoffs are considered, and the maturity of each technology is examined.

Chapter 4: Application integration

A J2EE application that is not integrated with its environment cannot do much. This chapter is about integrating your J2EE application with other applications and services using the Java APIs for XML. Proven approaches to J2EE systems integration and architectural patterns are presented. Traditional J2EE technical approaches to systems integration are compared to the new, XML-based approach.

This chapter details the creation and consumption of web services in J2EE, including discussions and examples of SOAP, UDDI, and WSDL. Producing, registering, and consuming web services in J2EE is demonstrated using the Java APIs for XML. This chapter also discusses possible integration issues with non-Java web service implementations, specifically Microsoft .NET.

Chapter 5: User interface development

This chapter discusses user interface development for a J2EE and XML application. The pure J2EE approach to user interface development has a number of limitations, including the mixture of presentation elements with application code and the inability to centrally manage application views in some circumstances. Recent developments in XML technology, including XSLT processing and web publishing frameworks have the potential to overcome these limitations.

In this chapter, we describe these two alternative XML presentation layer architectures and compare them to the pure J2EE approach. Detailed examples using XSLT and web publishing frameworks demonstrate how you might implement a multidevice, multilingual presentation layer for your J2EE application using XML technology to dynamically create user interfaces in various formats.

Chapter 6: Case study

This final chapter illustrates the use of the tools and techniques presented in previous chapters in the context of a simple, yet complete, case study. By providing an end-to-end example of a J2EE and XML solution, we further illustrate the feasibility and desirability of using XML in J2EE solutions.

You are guided through a brief development cycle from requirements and analysis to design and implementation. Along the way, the challenges faced are highlighted, and reasons behind key design decisions are articulated.

At the back

This book also contains three appendices on closely related topics. Appendix A contains a brief summary of the J2EE design patterns employed throughout the

book. Appendix B contains a tutorial on distributed system security concepts you should know before developing any J2EE solution. Appendix C provides a tutorial on the popular Ant build tool from the Apache Software Foundation.

Also at the back, you will find a helpful resources section, containing recommended books and web sites for learning more about the tools and standards discussed throughout the book.

Source code

The source code for all examples called out as listings in this book is freely available from the publisher's web site, <http://www.manning.com/gabrick>. The complete source code for the case study in chapter 6 is also available at the same address. Should errors be discovered after publication, all code updates will be made available via the Web.

Code conventions

`Courier` typeface is used to denote code, filenames, variables, Java classes, and other identifiers. **bold courier** typeface is used in some code listings to highlight important sections.

Code annotations accompany many segments of code. Certain annotations are marked with chronologically ordered bullets such as **❶**. These annotations have further explanations that follow the code.

about the authors

KURT GABRICK is a software architect and developer specializing in server-side Java technologies and distributed systems. He has designed and developed numerous systems using J2EE and XML technology for a diverse group of Fortune 1000 clients. Kurt has led various engineering efforts for software development and professional services firms. He currently resides in the Phoenix, AZ area, where he continues to code for fun and profit.

DAVE WEISS is an IT architect specializing in use case driven, object-oriented development with Java and XML. Dave has worked for multiple professional services companies, where he was responsible for software development methodology and training programs, as well as leading distributed systems development projects. Dave has authored numerous pieces of technical documentation and training materials. He currently resides in the San Francisco Bay area.

about the cover illustration

The figure on the cover of *J2EE and XML Development* is a man from a village in Abyssinia, today called Ethiopia. The illustration is taken from a Spanish compendium of regional dress customs first published in Madrid in 1799. The book's title page states:

Coleccion general de los Trages que usan actualmente todas las Naciones del Mundo desubierto, dibujados y grabados con la mayor exactitud por R.M.V.A.R. Obra muy util y en special para los que tienen la del viajero universal

Which we translate, as literally as possible, thus:

General collection of costumes currently used in the nations of the known world, designed and printed with great exactitude by R.M.V.A.R. This work is very useful especially for those who hold themselves to be universal travelers

Although nothing is known of the designers, engravers, and workers who colored this illustration by hand, the “exactitude” of their execution is evident in this drawing. The Abyssinian is just one of many figures in this colorful collection. Their diversity speaks vividly of the uniqueness and individuality of the world's towns and regions just 200 years ago. This was a time when the dress codes of two regions separated by a few dozen miles identified people uniquely as belonging to one or the other. The collection brings to life a sense of isolation and distance of that period—and of every other historic period except our own hyperkinetic present.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative and the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by the pictures from this collection.

To buy this book

author online

One of the advantages of buying a book published by Manning, is that you can participate in the Author Online forum. So, if you have a moment to spare, please visit us at <http://www.manning.com/gabrick>. There you can download the book's source code, communicate with the author, vent your criticism, share your ideas, or just hang out.

Manning's commitment to its readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the AO remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's web site as long as the book is in print.

To buy this book



Getting started



This chapter

- Describes important distributed systems concepts
- Discusses J2EE and formal development methodologies
- Identifies J2EE development tools and best practices
- Recommends J2EE testing and deployment strategies

This introductory chapter covers important concepts, tools, and techniques for building J2EE and XML applications. As a distributed application developer, you face a broad range of challenges as you begin each new project. These challenges range from architectural and design issues to tool selection and management of the development process. To overcome these challenges, you require both an appreciation for distributed systems development issues and knowledge of the specific tools that you can use in J2EE development.

Section 1.1 describes the aspects of distributed application development that you need to understand to make effective use of J2EE and XML. In that section we present the n-tier application architecture under which most enterprise Java systems are constructed today. We define the logical layers of these applications and describe the types of components and challenges associated with each layer. We also identify the specific types of challenges you are likely to face when designing your application and present alternatives for dealing with those challenges.

In section 1.1, we also cover the often-misunderstood area of distributed application security. Without the ability to secure your distributed application properly, its usefulness can quickly be negated. We summarize your options for securing communication channels and application components in this section.

Sections 1.2 and 1.3 describe the tools and techniques you need to have success with the J2EE platform. These range from defining an overall development process to choosing your design, development, and configuration management tools. We suggest popular open source tools, which are available for many aspects of development. We also suggest strategies for testing and deploying your J2EE and XML application.

1.1 *Distributed systems overview*

DEFINITION A *distributed computing system* is a collection of independent computer processes that communicate with one another by passing messages.

By the definition, every application or service you develop using J2EE and XML will be part of a distributed system. To build the best J2EE and XML solutions possible, understanding general distributed system concepts and design challenges is essential.

This section covers the subjects you need to know before worrying about how to integrate J2EE technology X with XML standard Y. Since we are

summarizing an entire branch of computer science in only a few pages, we strongly recommend the resources listed in the bibliography as further reading.

1.1.1 *Distributed systems concepts*

In the days of mainframe computing, processing was a centralized, closed, and expensive endeavor. Information was processed by large, costly machines and manipulated from the dreaded green-screen terminals that gave new meaning to the word dumb. Corporate, scientific, and governmental information was locked away in individual computing silos and replicated in various forms across all kinds of computer systems.

Mainframe computing is not all bad. The centralized model has enabled the construction of many high-performance, mission-critical applications. Those applications have usually been much easier to understand and implement than their distributed equivalents. They typically contain a single security domain to monitor, do not require a shared or public network to operate, and make any system crashes immediately obvious to both users and administrators.

Conversely, distributed applications are far more difficult to implement, manage, and secure. They exist for two primary reasons: to reduce operating costs and to enable information exchange. Distributed systems allow all types of organizations to share resources, integrate processes, and find new ways to generate revenue and reduce costs. For example, a supply chain application can automate and standardize the relationship between several organizations, thereby reducing interaction costs, decreasing processing time, and increasing throughput capacity.

In economic terms, distributed systems allow companies to achieve greater economies of scale and focus division of labor across industries. In business terms, companies can integrate entire supply chains and share valuable information with business partners at vastly reduced costs. In scientific terms, researchers can leverage one another's experience and collaborate like never before. And in technical terms, you have a lot of work to do.

What makes distributed systems so difficult to design and build is that they are not intuitive. As a human being, your life is both sequential and centralized. For example, you never arrive at work before getting out of bed in the morning, and when you do arrive, you are always the first to know. Distributed computing is not so straightforward. Things happen independently of one another, and there are few guarantees that they will occur in the right order or when they are supposed to. Processes, computers, and networks can crash at any time without warning. Designing a well-behaved, secure

distributed system therefore requires a methodical approach and appreciation of the challenges to be overcome along the way.

Distributed system components

At the highest level, a distributed system consists of four types of components, as depicted in figure 1.1.

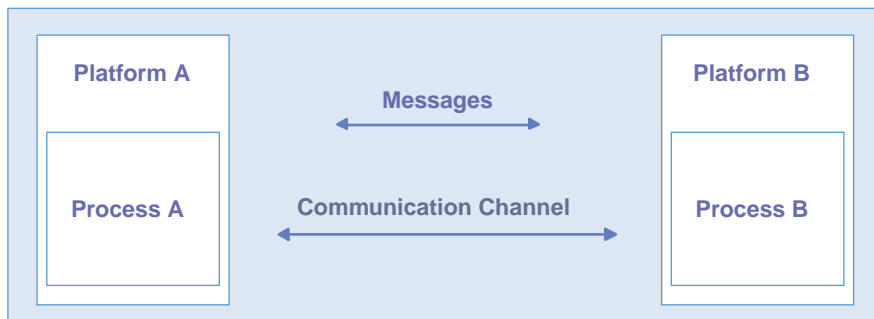


Figure 1.1 Distributed system components

- **Platforms**—Platforms are the individual computing environments in which programs execute. These can be heterogeneous hardware components, operating systems, and device drivers that system architects and developers must integrate into a seamless system.
- **Processes**—Processes are independent software components that collaborate with one another over channels. The terms *client*, *server*, *peer*, and *service* are often substituted for the term *process*, and each has a more specific meaning, as we discuss later in this section. Process can mean different things depending on the granularity with which one uses it. A process can represent an individual software object with a remote interface, a client or server that implements a particular protocol, some proprietary business application, or many other things.
- **Communication channels**—Communication channels are pipelines between processes that enable them to interact. The term usually refers to the computer network(s) that logically connect processes and physically connect platforms. Communication channels have both physical and logical aspects that are accounted for in any distributed system design.
- **Messages**—Messages are the data sent from one process to another over a communication channel. How these data flow between processes in a

reliable and secure manner is a question that requires much thought in the analysis and design stages of your development cycle. XML can facilitate defining both the semantics and processing of messages between systems, as we discuss in detail throughout the book.

The four types of distributed system components identified above are typically arranged in one of three distinct *architectures*, based on the ways in which individual processes interact with one another. These models are summarized in table 1.1.

DEFINITION *Distributed system architecture* is the arrangement of the software, hardware, and networking components of a distributed system in the most optimal manner possible. Creating distributed system architecture is a partly-science, mostly-art activity.

J2EE supports all the architectural models listed in table 1.1 to some extent, but is heavily focused on client/server architectures. Let us briefly examine each of these models. Table 1.1.

Table 1.1 Distributed system types

System architecture	Description
Client/server	A distributed interaction model in which processes do things for one another
Peer processing	A distributed interaction model in which processes do things together
Hybrid	A combination of client/server and peer processing models

The client/server model

Client/server is the architectural model of the World Wide Web, and the one with which you are probably most familiar. The client/server model is a distributed computing paradigm in which one process, often at the behest of an end user, makes a request of another process to perform some task. The process making the request is referred to as the *client*, and the process responding to the request is known as the *server*. The client sends a message to the server requesting some action. The server performs the requested action and returns a response message to the client, containing the processing results or providing the requested information. This is depicted in figure 1.2. This request-reply mechanism is a synchronous interaction model and is the basis of a family of higher-level interaction paradigms. For example, remote procedure calls

(RPC) and the Hypertext Transfer Protocol (HTTP) used on the World Wide Web both employ the client/server mechanism, but are quite different from each other at the application level.

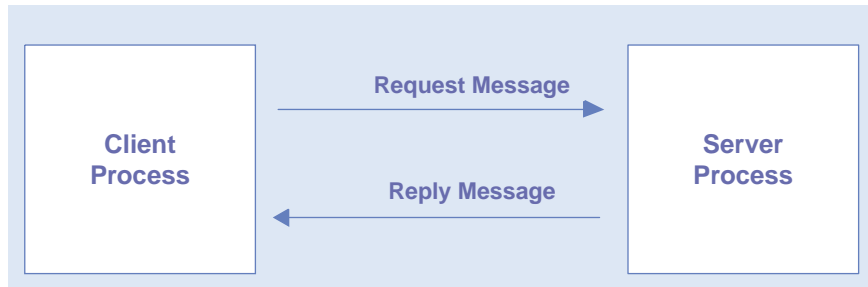


Figure 1.2 Client/server architecture

Client/server is a role-based model. The labels *client* and *server* only refer to a process's role in a specific interaction. In practice, one process may act as a client toward one process and as a server toward another. Two processes may also be servers to one another in different situations. Some of the possibilities for these relationships are illustrated in figure 1.3. The J2EE specification is currently focused on the server-side of this relationship through its endorsement of servlet, Java Server Pages (JSP), and Enterprise Java Beans (EJB) specifications.

Another important concept in client/server computing is *service architecture*. Servers usually provide a set of related functions and make them available to clients through standard interfaces and protocols. A common example is a Web server, which allows clients to send and receive resources in a variety of ways over the Internet via the HTTP protocol. While service architectures have been implemented in the past for things such as Web, mail, and DNS services, they are just beginning to take hold in business applications. In chapter 4, we discuss something called the *web services architecture*, the latest incarnation of the services architecture concept.

A set of related functions provided by a single server is a *service*. By encapsulating a set of related server functions into a service with a standard interface, the manner in which the service is implemented becomes irrelevant to the client. Multiple server processes are then dedicated to performing the same service for clients transparently. This is an essential technique employed commonly to provide fault tolerance, hide implementation details, and enhance performance in distributed systems. This is depicted in figure 1.4.

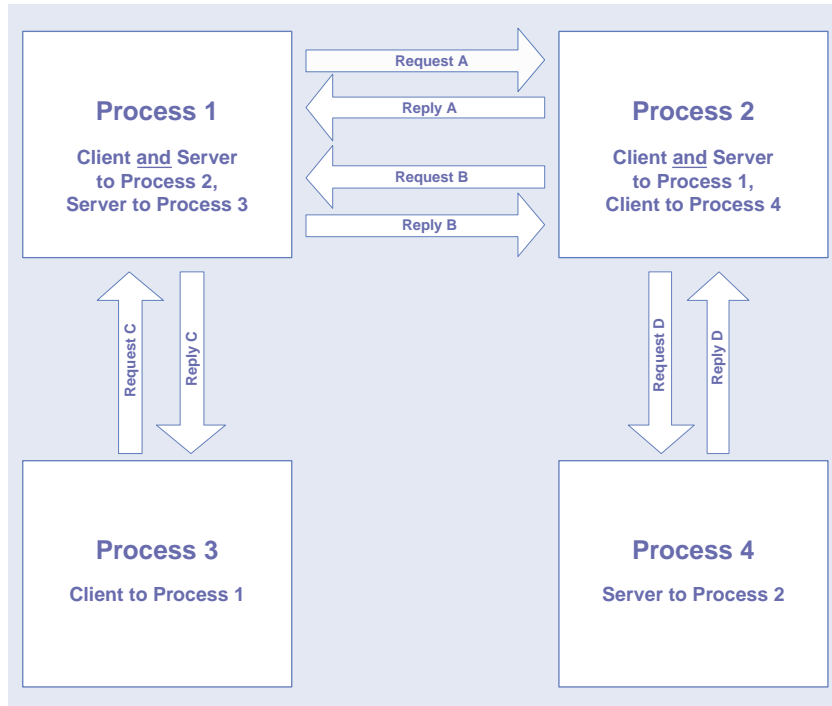


Figure 1.3 Role Playing in client/server systems

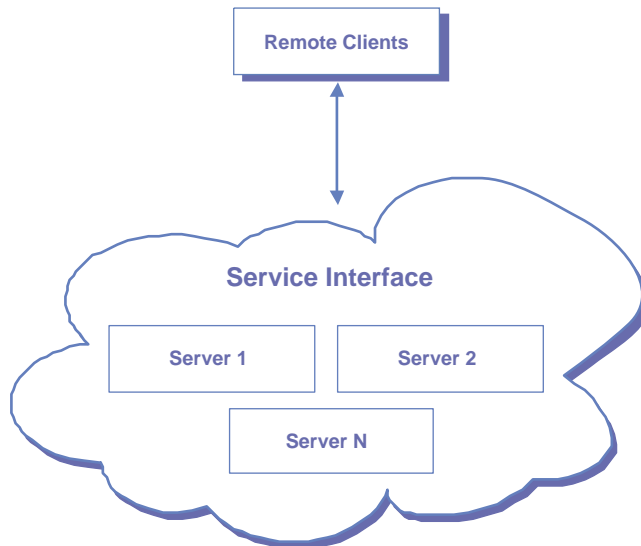


Figure 1.4 Service architecture concepts

To buy this book

J2EE is heavily focused on server-side, Web-enabled applications. This does not mean that other types of applications cannot be built using the J2EE platform, but does make creating Web-based, thin-client applications the most logical choice for most J2EE developers.

In chapter 3, we examine the client/server interactions that occur locally, inside your application. Chapter 4 describes client/server interactions between your application and other systems, including web services architecture. Finally, in chapter 5, we examine the client/server capabilities of J2EE in terms of user interfaces

The peer model

In this architectural model, independent processes collaborate to perform some task in a coordinated fashion. The peer approach is common in situations where either a lot of computing power is needed to perform an intense calculation or where independent systems need to guarantee that synchronized states are maintained. Any system that is *transactional* makes use of this model for at least part of its functionality.

The peer model treats all processes as equals, although it often requires one of them to act as a group coordinator. An example of a peer processing situation in scientific applications is gene sequencing; a business processing example is executing a distributed purchasing transaction. In these situations, each process calculates part of some result and contributes it to the whole. For example, as a customer places an order, a pricing process calculates a specific customer's price for each item on an order and adds those prices to the order information.

J2EE supports peer processing via the Java Transaction Architecture (JTA) API. Using this API, your components can interact in a scoped, coordinated way with each other and with external systems. JTA is one of the many J2EE APIs available to you, and transactional support is one of the key features of any J2EE EJB container.

Merging the client/server and peer models

There is no reason the client/server and peer models cannot coexist in the same system. In practice, most substantial systems manifest traits of both the client/server and peer processing models. For example, figure 1.5 shows a web client invoking an e-commerce service provided by a merchant server to place an order for a product. The e-commerce server accepts the request and connects to the back-end fulfillment system as a client. The fulfillment system in turn collaborates with the pricing and inventory systems to complete the

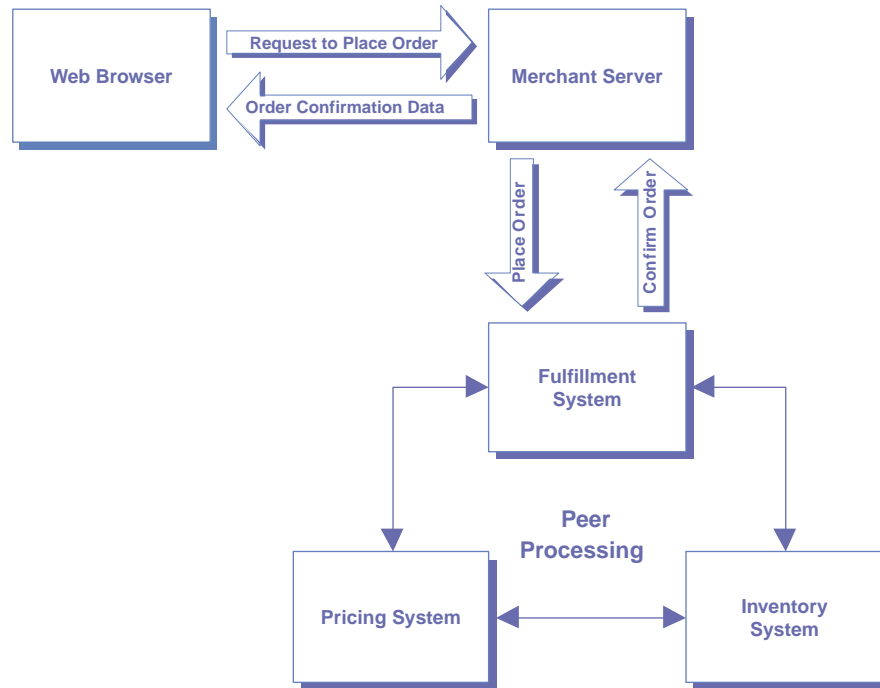


Figure 1.5 Combining client/server and peer processing architectures

order and generate a confirmation number. This number and other order data are then sent back to the original client process via the merchant server.

The hybrid model demonstrated here is used frequently in business application development. Chances are good that you will use it in your J2EE development projects rather than using either client/server or peer processing exclusively.

Distributed system software layers

Client/server and peer processing architectures rely heavily on the layered approach to software development depicted in figure 1.6. All processes, whether acting as server, client, or peer, must execute on a computer somewhere. Each computer consists of a specific operating system and a set of device drivers, all of which come in numerous varieties. Since it would be foolish to try to predict every operating environment in which a process may be required to run in overtime, a mechanism is needed to divorce the process from its execution environment. And so was born a new class of software product, called *middleware*.

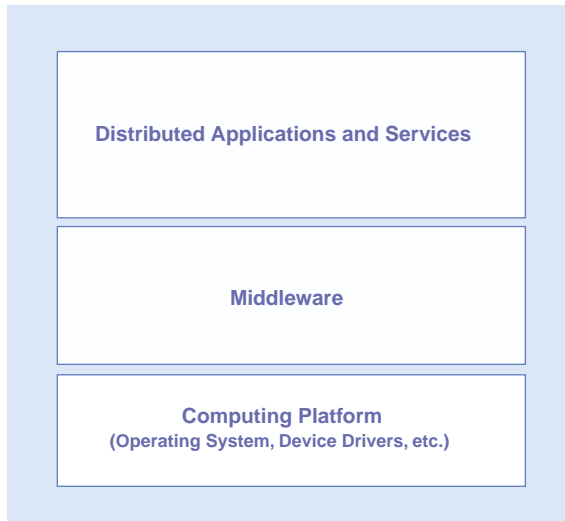


Figure 1.6
Distributed system
software layers

Middleware, such as the J2EE products discussed in this book, exists to overcome the differences between different computing platforms. It exposes a common set of services across platforms and provides a homogeneous computing environment in which distributed applications can be built. Software that relies solely on its middleware environment can be deployed on any platform to which the middleware has been ported. And since distributed systems must grow incrementally over a period of time in different financial, political, and business environments, the ability to run on a wide variety of platforms is crucial to the longevity of most systems. Middleware is an essential ingredient in distributed systems development.

One of the most powerful aspects of J2EE is the broad range of middleware services it provides to developers. The set of service APIs that are currently a part of the J2EE specification is summarized in table 1.2. As you see, J2EE provides built-in support for publishing and locating resources, asynchronous messaging, transactions, and a host of other services. If you have worked with J2EE in the past, you are probably familiar with many of these. One API that is of particular interest to us in this book is JAXP, which we discuss in detail in the next chapter. You will also see XML as middleware for your data throughout the remaining chapters.

Table 1.2 J2EE middleware services

Enterprise Java API	Application in J2EE
Java Naming and Directory Services (JNDI)	Provides a standard mechanism for locating resources, including remote objects, environment properties, and directory services.
Java Database Connectivity (JDBC)	Provides vendor-neutral access to enterprise relational database management systems.
Java Message Service (JMS)	Provides reliable point-to-point and publish/subscribe messaging for J2EE components.
Java Transaction API (JTA)	Provides mechanisms for declaring, accessing, and coordinating transactional processing.
JavaMail	Provides support for sending Internet email from J2EE applications.
Java Activation Framework (JAF)	A mechanism of inspecting arbitrary data and instantiating objects to process it, required by the JavaMail API.
Java API for XML Parsing (JAXP)	Provides basic support for XML access from Java and a service provider interface for parsers and transformers.
J2EE Connector Architecture	An architectural framework for plugging vendor-supplied resource drivers into the J2EE environment.
Java Authentication and Authorization Service (JAAS)	Provides basic mechanisms for authenticating users and authorizing their access to resources. This API is being integrated into the base Java platform, version 1.4. At the time of this writing, the J2EE specification still explicitly references it as a required service.

At the top of the distributed software stack are distributed applications and services. These fall in the realm of the business application developer, and are probably the part of distributed systems development in which you are most interested. The distinction between applications and services made in figure 1.6 illustrates that not everything built in a distributed environment may be a full-fledged application.

DEFINITION An *application* is a logically complete set of functions that may make use of a number of services to automate some business or other human process.

To illustrate this point, an e-commerce shopping site can be seen as an application with various features, such as searching, purchasing, and order history retrieval. A server implementing the file transfer protocol (FTP) is just a service that allows users to upload and download arbitrary files.

Whether you are building a service or an application has a dramatic effect on the activities you undertake and the considerations you need to make during analysis and design. However, distributed services and applications do share enough characteristics that we usually discuss their properties together. The distinction between the two becomes important in chapter 4, where we look at integrating external services into your applications.

DEFINITION A *service* is a general set of functions that can be used in various ways by specialized applications. Services usually only have one primary function, like locating resources or printing documents.

1.1.2 *N-tier application architecture*

Many distributed application architects find it useful to group their development tasks in terms of logical *layers*, or *tiers*.

DEFINITION An *application layer* is a logical grouping of system components by the functionality they provide to users and other application subsystems.

In general, every distributed application does similar things. It operates on its own data, interacts with external systems, and provides an interface to its users. This general pattern gives rise to the *n-tier architecture* depicted in figure 1.7.

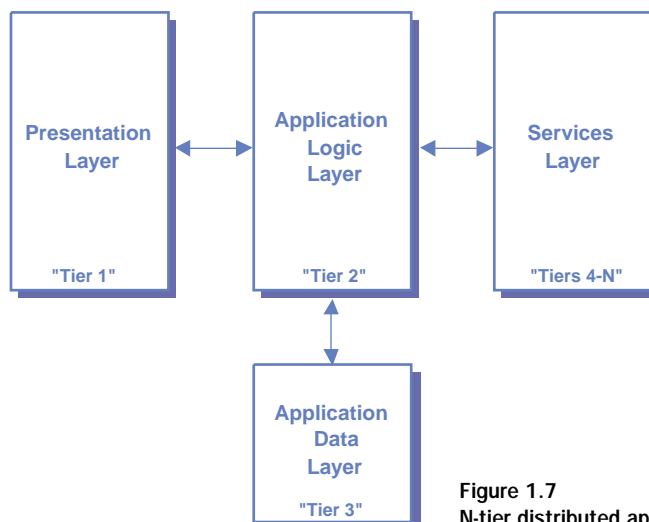


Figure 1.7
N-tier distributed application architecture

The presentation layer

The presentation layer refers to those components responsible for creating and managing an application's interface(s) with its users. Technologies employed here include web servers, dynamic template processing engines, and network-aware client applications such as web browsers. In J2EE, presentation layer components include servlets and Java Server Pages (JSP) running in the J2EE web container.

The primary challenge at this layer of the architecture is the creation and management of different, synchronized views of the application for different users, based on access rights, client-side rendering capabilities, and other factors. Building a presentation layer that is robust and manageable is not easy. We take a detailed look at how this can be done using a combination of J2EE and XML technologies in chapter 5.

The application logic layer

The application logic layer (known as the *business logic layer* to business application developers) refers to the components responsible for implementing the functionality of an application. These components must manage the application's data and state while performing the specific operations supported by the application. In J2EE, application logic is usually implemented by Enterprise JavaBeans (EJB) running in the J2EE EJB container. Components at this layer implement the resource-intensive, often transactional portion of your application.

Challenges at this layer involve ensuring correct behavior and data integrity, interactions between system components, error handling, and performance optimization. Building a flexible, high performance application logic layer is quite challenging. We examine the ways in which XML might help J2EE developers do this in chapter 3.

The application data layer

This layer refers to the components that manage an application's own, internal data. In J2EE, these data are typically under the direct control of a relational database management system (RDBMS) like Oracle Enterprise Server or IBM DB/2. J2EE now mandates the presence of an RDBMS in its server environment. In some situations, you may not need to write components to directly interact with a data store. If all your data-aware objects are EJB Entity Beans that employ Container Managed Persistence (CMP), the EJB container handles all database interaction on your behalf. This, of course, comes at the price of extra configuration and a loss of flexibility in your data and/or object models.

Challenges at this layer include effective use of system resources, database connection pooling, and performance optimization. The EJB container and JDBC driver classes handle most of this for you in J2EE, but an RDBMS may not be the right place to store your data in some circumstances. We examine such situations in our discussion of XML at the J2EE application layer in chapter 3.

The services layer

The services layer refers to an application's external environment, with which it collaborates in a variety of ways. A distributed application that does not touch external systems is rarely useful. The services layer accounts for tiers four through n of an n-tier application, since services can use other services and there is no theoretical limit to the number or variety of relationships between systems.

As the developer of a specific application, the challenge at this layer is how to interact with the environment in the most effective way. Chapter 4 discusses this layer in detail and provides useful architectural patterns and techniques for integrating remote services into your J2EE-XML application. It explains your application integration options and covers the latest developments in this area from a J2EE and XML developer's perspective.

1.1.3 *Overcoming common challenges*

Since all distributed systems share some basic characteristics, they also have some challenges in common. In this section, we examine common issues faced by every distributed system architect, as well as the strategies and design goals frequently employed to overcome them.

Heterogeneity of system components

Computer hardware and software comes in seemingly infinite varieties, and you never find two components from different vendors that are exactly alike. This is true for computers, networks, and software products, as well as the applications built on top of them. The nature of a distributed system prevents us from making bold predictions about when and how various services and applications are going to be implemented, where they will need to run, or how they will need to be extended. After all, a key benefit of the distributed model is that the system can grow incrementally over time.

There are two primary strategies you can employ to overcome the problem of heterogeneity. The first is to abstract the differences in computing environments by using middleware, as described in section 1.1.1. This enables you to write more general applications and services that can be deployed to many

different environments over time. Your ability to move code and processes from one location to another is limited only by the capabilities of your middleware and the platforms it supports.

The second strategy is to abstract differences in communication channels and data representations through use of standards and protocols. For instance, the Internet is a collection of disparate computers and networks that are able to collaborate only because they speak the same languages. Separating the application-level and transport-level communication aspects is the key. To do this, protocols and data formats must be agreed to and, in the case of the Internet, widely accepted.

Evidence of this strategy's success can be seen in many Internet services, including the World Wide Web and Internet email. This concept is currently being extended to standardize business communication over the Internet using XML technology. We discuss this topic in detail in chapter 4.

Flexibility and extensibility

Shortsighted is the architect who believes he can predict the future requirements placed on his or her system. The migration path from a closed e-commerce site to an integrated supply chain is far shorter in the business world than it is in the technical one. A key design goal for all distributed systems is to maximize system flexibility and make extending functionality as painless as possible. Unfortunately, it is difficult to mandate the ways in which this is accomplished.

One way to face this challenge is to do a good object-oriented analysis of your functional requirements. Study each requirement intently and try to abstract it into a more general class of problem. Then, by building functionality that addresses the more general class of problem, your system will be better prepared to handle modifications and extensions to its capabilities in the future. When functionality needs to be changed or extended, you will be able to reuse existing components rather than building from scratch.

For example, just because your company repairs vacuum cleaners does not mean that you build a vacuum cleaner tracking system. You build a workflow engine that can track the states of things, send notifications, and route messages. Then you apply your engine to the task of tracking vacuum cleaner repair jobs. And next month, when your company expands into toasters and microwave ovens, you go on vacation to reflect on your genius.

This book discusses numerous strategies you can implement with J2EE and XML to generalize your system and maximize its flexibility. In chapter 3, we take a general approach to creating interfaces between components. In chapter 4, we discuss general mechanisms for integrating your application with its

environment. In chapter 5, we take a general approach to serving views of your J2EE application over the Web.

Vendor independence

Your system does not exist in a vacuum. Hardware, operating systems, middleware, and networking products all play a role both in enabling and limiting the capabilities of your system. A well-designed system is one that operates in the context of hardware and software vendor implementations, but is not tied to it.

DEFINITION An *open system* is one in which components can be implemented in different ways and executed in a variety of environments.

If your system is really open, the decisions made by your product vendors are much less of a threat to it over time. This can be essential to the longevity of your system and your reputation as its creator.

Addressing the issue of vendor independence is a two-step process. First, you must find vendors who conform to industry-supported standards whenever possible. Is it safer in the long-term to implement a web site in a proprietary scripting language provided by one vendor, or to implement it in Java Server Pages? Since you are reading this book, we hope the answer is obvious.

The second step is far more crucial. You should make proprietary extensions to standard mechanisms only when absolutely necessary. In such cases, going with a vendor's solution is probably better than inventing your own, because, you hope and expect, they did a lot of thinking about it first. For example, J2EE does not currently address logging requirements, although it will soon. To implement logging in your components, you can either use an existing logging API or create your own. It is probably more expeditious to use what is already available. However, you should always wrap any vendor-specific code in a helper class and access it via a generic interface. That way, changing from the proprietary mechanism to the standard one will be much easier in the future. The Façade design pattern is a useful approach. See the bibliography for a list of references on design patterns if you are unfamiliar with this pattern.

Embracing proprietary extensions should be avoided whenever possible. The more you do this, no matter how convenient it makes your short-term plans, the more married you are to your original implementation and the more long-term risk there is to the system.

Scalability and performance

Most system stakeholders want to believe that system use will grow exponentially over time as more business relationships are solidified and users begin to see the subtle genius of the concept. Whether this is true is irrelevant. The danger is that it could be true. And as demand for system resources increases, supply must also increase without negatively impacting performance. Your solution must be *scalable*.

DEFINITION *Scalability* is a measure of the extent to which system usage can increase without negatively impacting its performance.

Every system must deal with the good-and-evil struggle between functionality and performance. The more functionality the system provides, the more time and resources are needed to provide it. The slower the system is, the less likely it is to be used.

There are several ways to deal with performance concerns. One way is to eliminate functionality. If your boss will let you do this, please email us so we can come work with you! Another way is to streamline functionality wherever possible. For example, make communication between processes asynchronous whenever possible, so execution threads do not block while interacting with remote systems. Ensuring that your distributed algorithms are streamlined and that time-sensitive processing has few external dependencies can be half the battle in performance tuning.

Assuming your system is fine-tuned, throughput can be enhanced using replication, load balancing, proxying, and caching.

DEFINITION *Replication* is the dedication of additional hardware and software to a given activity in order to provide more processing capability.

Combining replication and load balancing is sometimes referred to as *server clustering*. Setting up *proxies* and caching data can be even better than replicating functionality and balancing loads.

DEFINITION *Load balancing* is the distribution of demand for a service across all servers that provide the service, ensuring that available resources are being used evenly and effectively.

DEFINITION *Caching* is the technique of storing processed data so your servers will not need to regenerate a set of data that has not changed since the last time it was requested.

Caching proxy servers can be used to intercept requests for resources, validate them before passing them on, and often even returned cached data to clients themselves. Unfortunately, caching and proxying can't be used in update requests, which limits their use to the retrieval of existing data.

The leading J2EE server providers offer scalability in different ways, but all provide some level of server clustering and load balancing. If your provider cannot make your J2EE environment scale, change providers. Scalability and other nonfunctional enhancements are severely lacking in J2EE, but most enterprise-level vendors have found ways to pick up the slack for now.

Concurrency and correctness

Providing reliability is not just a matter of ensuring that the system does not crash. An equal measurement of your system's reliability is the extent to which it operates consistently. Regardless of load, time of day, and other factors, your system must always keep itself in a valid state and behave in a predictable way. The integrity of your system's data is not hard to achieve in most distributed applications, because they rely at some point on a database management system (DBMS) that guarantees such integrity. The state and behavior of a running application, however, is the responsibility of its designer and developers.

Ensuring that any logic-intensive application will run correctly in all situations is a complicated task. In a distributed system, it is even more so. This is because servers in distributed systems must provide access to shared resources to various clients, often concurrently. It is the responsibility of each service implementer to ensure that information updates are coordinated and synchronized across all client invocations. To address this, each distributed component should have a detailed state model and be tested thoroughly. Assume nothing works properly until proven otherwise. You will thank yourself when your system goes live and you still have your weekends.

Ensuring that individual J2EE components work together like they should can be achieved by using the aforementioned JTA API and the transactional capabilities of the EJB container. Your application can also lean on the transactional capabilities of its relational database in some situations.

Error handling

Dealing with error conditions in distributed systems is a real challenge. This is because the failures that occur do not crash the entire system. A part of the system fails, and it is up to the other components to detect the failure and take appropriate action. And since certain types of failures can't be detected easily or at all, individual components need to be overly suspicious of errors when interacting with each other.

There are various types of distributed system failures, which can be grouped as follows:

- *Process failures*—These are failures of individual processes. They can be further classified, based on whether or not the failure can be detected by other processes when it occurs.
- *Omission failures*—These are failures in communications, and include partial message transmissions and corruption of messages during transport.
- *Arbitrary failures*—These are random failures or unpredictable behavior. This is the worst kind of failure, and the hardest to predict and guard against.

Once an error has been detected, there are a couple of ways to try to recover from it. In the case of a communication problem, a dropped or corrupted message can be resent. This is the technique employed by the Simple Mail Transport Protocol (SMTP) used by email systems on the Internet. To deal with a processing failure, the original service request can be redirected to another server. This technique is known as *fail-over* and can be initiated explicitly by the client or by the service.

Fault tolerance is a key measure of system reliability. This term refers to the degree to which your system can detect and recover from the independent failures of its components. This is accomplished by fault-masking techniques as described above. *Fault masking* simply means hiding errors from system clients.

Your J2EE provider should provide some fail-over mechanism as part of its server clustering functionality. Still, it will be the responsibility of your application components to detect any application-level failures and recover from them (by masking them) whenever possible. Try to be specific in terms of exception throwing and handling in your code. It is easier to recover from an exception if you know specifically what it is and how it happened when you catch it. We have seen many components that feature epic try blocks and only catch `java.lang.Exception` or `java.lang.Throwable`. If your code does not observe exceptions closely, its chances of masking them are quite slim.

Transparency

Transparency in its many forms is a design goal that can make your system easier to use and more flexible. The principle is that the distributed nature of the system should be transparent to its users as well as to developers of individual applications. This is done to maximize the scalability and flexibility of the system. There are various types of transparency, as summarized in table 1.3.

Table 1.3 Transparency types in distributed systems

Transparency type	Description
Network transparency	All resources are accessed in the same manner, regardless of their actual location on the network.
Location transparency	The amount of hardware and software resources dedicated to an activity can be increased without affecting clients. This enables the system to scale more easily.
Failure transparency	Through fault handling techniques, the system allows clients to complete their tasks despite hardware and software failures.
Mobility transparency	Resources in the system can be rearranged without affecting users.

Using naming services to locate resources and leveraging remote object architectures are two ways in which you can enable network and mobility transparency in your application. The Java Naming and Directory Interface (JNDI) and Remote Method Invocation (RMI) support this type of transparency in J2EE. Your J2EE server provider usually provides location transparency as part of server clustering. As noted in the previous section, you must share responsibility for failure transparency with your server vendor.

System security

Distributed systems exist to share valuable resources among specific parties. Take pains to ensure that these resources are not shared with or modified by anyone else. Finding ways to share information securely over communication channels is the primary challenge of security. There are two main aspects to security in distributed systems. One involves verifying the identity and access rights of each user. We will discuss that topic here. The other involves the broader topic of protecting the application from hackers and other would-be users who should not have any access to the system. More information on that topic can be found in appendix B.

The first critical step in securing your system is having a reliable authentication and authorization system for its intended users.

DEFINITION *Authentication* is the process of verifying that someone is who he or she purports to be.

J2EE addresses authentication and authorization via the Java Authentication and Authorization Service (JAAS). This is an implementation of the Pluggable Authentication Module (PAM) security architecture, in which various security provider implementations can be plugged in to your J2EE environment. Each of these providers might implement authentication and authorization in different ways, but your components are shielded from the details and always access security information through a standard interface.

DEFINITION *Authorization* is the process of ensuring that each authenticated user can only access the resources that he or she has the right to access.

JAAS is soon to become a part of the base Java platform, in version 1.4. Using JAAS may seem like an obvious way to go with J2EE security requirements. The devil can be found in the details, as usual. There are currently two major drawbacks to using JAAS. The first is that you must declare your application security policy in deployment descriptors and configuration files rather than within the application itself. This can be error-prone and inconvenient, especially in the case of web applications. It is often impractical to rely on your J2EE container to authenticate and authorize users, especially when they register and self-administer their accounts via the Web. If your security policy must be updated dynamically at runtime, using JAAS can be impractical. Your application security model must also fit well with such JAAS concepts as authorization realms and principals.

The second drawback is the naive simplicity of many JAAS provider implementations. The out-of-the-box JAAS provider usually consists of authorization realm and credential information being stored in a plain text file or unencrypted database fields. This means that, even if you find a way to delegate your application security to the container, the manner in which your application is secured is very suspect.

The solution to both these problems is to find or develop a JAAS module that integrates well with your application object, data, and security models. Being able to map container-understood values to meaningful application data is the key. If this cannot be done, using container-level security can be problematic. We have not seen any implementations that do this well, but remain hopeful that such advances will be developed.

1.2 *The J2EE development process*

Implementing a complex software system is all about managing complexity, eliminating redundant efforts, and utilizing development resources effectively. This is especially true in the J2EE environment, where you are building an n-tier, distributed system. Determining what process you will follow to complete your application on time and on budget is the first critical step on the path to success. You must then determine which tools to use and how to use them to support your development process. Because these decisions are so critical, this section provides an overview of some of the most popular development methodologies and tools used on J2EE projects.

1.2.1 *J2EE and development methodologies*

Numerous development methodologies exist for object-oriented projects, and choosing one to adopt can be difficult.

DEFINITION A *development methodology* defines a process for building software, including the steps to be taken and the roles to be played by project team members.

For component-based development with J2EE and XML, finding one that exactly fits your needs is even more challenging. This is true because most development methodologies are robust project management frameworks, generically designed to aid in the development of software systems from the ground up. J2EE development is about implementing applications in an existing middleware environment, and the detailed, complicated processes prescribed by most methodologies can be partly inapplicable or simply too cumbersome to be useful on J2EE projects.

An example of this is the Rational Unified Process (RUP), developed by the masterminds at Rational Software. RUP provides a detailed process for object-oriented development, defining a complicated web of processes, activities, and tasks to be undertaken by team members in clearly defined roles. While this sort of methodology can be useful and necessary when building and maintaining, say, an air traffic control system, it is impractical to implement on a short-term, J2EE development project. J2EE projects usually feature a handful of developers tasked with building a business application that needs to be done some time yesterday. If, on the other hand, you are developing a complicated

system over a longer timeframe, RUP may be right for you. You can get information on RUP at <http://www.rational.com>.

While some methodologies are too thick for J2EE, others can be too thin. A methodology that does not produce enough relevant artifacts (such as a design) can be easily abused and its usefulness invalidated. The best, recent example of this is eXtreme Programming (XP), a lightweight methodology championed by many industry luminaries of late. XP is the ultimate methodology for hackers. It is extremely fluid and revolves almost exclusively around code. The XP process goes from requirements gathering to coding test cases to coding functionality. The number of user stories (in XP parlance) implemented and the percentage of test cases running successfully at the moment are the measure of success. You can get more information on XP at <http://www.extremeprogramming.org>.

XP is a lightweight, dynamic methodology, easily abused and often not appropriate for large development projects. One concern with XP is that it does not produce sufficient analysis and design documentation, which can be essential in the ongoing maintenance of a system, including training activities. J2EE development projects usually consist of small teams building functionality in the context of rapidly changing requirements. XP can provide benefits in the areas of quality assurance and risk mitigation under such circumstances. However, be cognizant of potential longer-term issues surrounding the architecture of your system and the lack of design documentation over time.

The trick to reaping the benefits of a methodology in J2EE is finding the right mix of tools and techniques that will enable your team to execute with more predictable results and higher quality. Methodology is only useful to the extent that it makes your product better. So, rather than choosing an existing, formal methodology, you may choose to roll your own, using the principles upon which most modern methodologies are based. These common principles are summarized in table 1.4.

Table 1.4 Common object-oriented development methodology principles

Principle	Description
User driven design	Software should be developed to satisfy the concrete requirements of its users. It should function in the way users would like to use it. Potential future requirements should be analyzed, but functionality that does not satisfy a known requirement need not be developed just in case.

(continued on next page)

Table 1.4 Common object-oriented development methodology principles (*continued*)

Principle	Description
Iterative, incremental development	A software development release should be accomplished using several iterations of the development process. Each iteration cycle should be short and small in scope, building upon any previous iteration by an increment. This enables the modification/clarification of requirements, enhancement to the design, and code refactoring during the development phase.
Risk mitigation	The most technically risky aspects of the system should be developed first, providing validation of the overall architecture and finding problems as quickly as possible.
Quality assurance	Testing must be an integral part of the development process, and a problem tracking/resolution process must be used and managed.

1.2.2 J2EE development tools

Choosing the right set of analysis, design, and development tools can greatly enhance the productivity of your team and the effectiveness of your processes. The ideal set of tools you should have for a J2EE build can be summarized as follows:

- *Analysis and design tool*—A visual drawing environment in which you can model your system, developing various UML diagrams that describe aspects of it.
- *Development tool*—Also known as an *integrated development environment* (IDE). While not required, an IDE can speed development time greatly. This is especially true when developing thick-client applications.
- *Build tool*—A utility to manage your development configuration and enable autodeployment of your components to the J2EE environment. Certain IDE products perform this function for certain server environments.
- *Source code control tool*—A shared repository for your code base in various versions of development.
- *Testing tool(s)*—Utilities to perform various types of testing on your components. We examine the complicated area of testing in section 1.3.
- *Problem tracking tool*—An often-missing component that integrates with your source code control environment to track problems from identification to resolution.

We present some common choices for each of these tool groups, along with their respective strengths and weaknesses, in the remainder of this section.

Analysis and design tools

Two of the most common choices in this area are Rational Rose by Rational Software and Together Control Center by TogetherSoft Corporation. Rational Rose is the old-timer of the two, written in native code for Windows. Together Control Center is a Java-based newcomer that is taking the industry by storm. Discovering which tool is right for you will depend on how you plan to model your system and translate that model into code.

Being a Windows application, Rational Rose's user interface is quite intuitive and does things like drag-and-drop operations quite well. Rose is an excellent tool for diagramming at both the conceptual (analysis) and design levels. It is not a real-time modeling environment, meaning that you must explicitly choose to generate code from your diagrams when desired. This is a good thing when working at the conceptual level, when the classes you create do not necessarily map to the implementation classes you will build. Also, the code generated from Rose is notoriously bloated with generated symbols in comments. Rose can be quite unforgiving when its generated tags have been modified and you attempt to do round-trip engineering.

Together Control Center, on the other hand, is a Java-based tool that still suffers from symptoms of the Java GUI libraries. It is not as intuitive to diagram with, requires a healthy chunk of memory, and can have some repainting issues from time to time. On the other hand, it is a real-time design and development environment. As you change a class diagram, the underlying source files are updated automatically. The reverse is also true. This makes the product a wonderful tool for low-level modeling and can even be a complete development environment when properly configured. For conceptual modeling it is less effective, since it assumes that any class you represent must be represented in code.

So the selection criteria between these tools is about the level(s) at which you intend to model, to what extent you plan to do round-trip or real-time engineering, and of course the price you are willing to pay. Both products are abhorrently expensive in our opinion, but that is all we will say on the matter. There are other UML tools around with smaller feature sets and user bases that you should explore if pricing makes either of these two impractical for your project.

Development tools

If you are developing in a Windows environment, the decision concerning the use of an IDE should be based on several criteria. First, is there an IDE that integrates well with your chosen J2EE server? What will it buy you in terms of

automating deployment tasks? Second, does your team share expertise in a particular IDE already? Third, are you doing any thick-client development that requires a visual environment? The answers to these questions should point you in the direction of a particular IDE.

Three of the most common commercial IDEs used on J2EE projects are WebGain Studio (which includes Visual Caf), Borland's JBuilder, and IBM's Visual Age. WebGain Studio is a complete J2EE development environment that integrates best with BEA System's WebLogic application server. Visual Age is the obvious choice for development on IBM's WebSphere platform. If you have already decided on a commercial J2EE vendor, the best IDE to use is usually quite obvious. If you are using an open source server like JBoss or Enhydra, the most important feature of an IDE may be its ability to integrate with the Ant build tool described in the next section. Ant integration is currently available for Visual Age, JBuilder, and the NetBeans open source IDE.

Build tools

Whether or not you choose to use an IDE, your project is likely to benefit from an automated build utility. Deploying J2EE components into their run-time environment involves compiling the components and their related classes, creating deployment descriptors, and packaging the components into JAR, WAR, or EAR files. All these files must have very specific structures and contents and be placed in a location accessible to the server. This whole packaging and deployment process is a complicated configuration task that lends itself to the use of a build tool. The build tool can be configured once and automate the build and deployment process for the lifetime of the component(s).

The most significant and recent development in this area is the Ant build tool, part of the Apache Software Foundation's Jakarta open source effort. Ant is a platform-independent "make" utility that uses an XML configuration file to execute tasks and build targets. Ant has a set of built-in tasks that perform common functions, such as compiling source files, invoking the JAR utility, and moving files. There are also a number of custom tasks available that extend Ant to provide specific types of functionality, such as creating EJB home and remote interfaces from an implementation source file and validating XML documents. One of the nicest things about Ant is its portability between operating systems. Your properly defined project will build on Windows and UNIX systems with very minor, if any, modifications to it.

For a brief introduction and tutorial on Ant, please refer to appendix C. The latest information about Ant can be found at <http://jakarta.apache.org>.

Source code control tools

J2EE applications are almost always developed in a team environment. Team development requires a source code repository and versioning system to manage the shared code base during development. The Concurrent Versioning System (CVS) is an open source versioning system used widely throughout the industry. It is available for UNIX and Windows environments, and provides enough functionality to meet the needs of most J2EE development teams. More information about CVS can be found at <http://www.cvshome.org>.

Teams needing tools that have vendor support or which are integrated into a particular IDE or methodology could choose a commercial tool instead. Such leading tools include Rational Software's Clear Case and Microsoft's Visual Source Safe. Another consideration in choosing a source control tool may be how you plan to implement problem tracking and integrate it with the management of your code base. More important than which tool you implement is the mere fact that you have one and use it.

Testing tools

Table 1.5 displays the major categories of testing that can be performed on your J2EE application and the intended goals of each. Note that the various testing types are known by different names to different people. This list is only a stake in the ground to frame our discussion of testing in section 1.3.

Table 1.5 Software testing types

Testing type	Description
Unit/functional testing	This refers to testing individual software components to ensure proper behavior. The developer usually performs this activity as part of the development process.
System testing	This usually refers to testing the functionality of the entire application, including the interactions between components and subsystems. Often, external integration points are simulated using test harnesses to control the tests.
Integration testing	Integration testing involves testing the functionality of the interaction between your application and external systems, including the proper handling of security and failure conditions.
Performance/load testing	This involves simulating heavy use of the application by clients to determine scalability and discover potential bottlenecks.
User acceptance testing (UAT)	This involves getting real users to try the system, examine its functionality, and report gaps between functionality delivered and original requirements.

There are many options for each type of testing you need to perform. Many of the best tools available in the area of unit testing in Java are open source tools. JUnit is a popular open source testing package for Java components. Using this package, you write test cases in Java and add them to your suite of unit tests. The framework then runs your tests and reports statistics and error information. Information about JUnit can be found at <http://www.junit.org>.

JUnit has also been extended to do more specific types of testing as well. For automated Web testing there is HTTPUnit. For server-side J2EE testing there are JUnitEE and Apache's Cactusproject. Information about HTTPUnit and JUnitEE can be found on the JUnit site listed above. Information about Cactus is at <http://jakarta.apache.org>.

For performance testing, you may choose to purchase a commercial product that can simulate heavy usage of your applications. Some vendors in this area offer a testing product to purchase and will also test and monitor your application's performance over the Internet for you. If you are in need of such services, you may want to investigate the Mercury Interactive testing tools at <http://www.mercuryinteractive.com>.

Problem tracking tools

The tool you use to track application errors during (and after?) testing is an important component of your development process. Software developers often struggle with what to do once a problem has been discovered. The bug must be identified, documented, and reproduced several times. Then it must be assigned to a developer for resolution and tracked until it has been fixed. Seems simple, but the implementation of this process is often overly complicated and mismanaged.

Teams usually implement problem tracking in very nonstandardized and problematic ways. Emails, spreadsheets, and MS Access databases are not uncommon implementations of *bug logs*. Many development projects use a bug tracking database, usually written in-house by a college intern with limited skills. These one-off tracking mechanisms suffer because they do not feature a notification system and are often improperly used by testers, project managers, and developers.

To generalize a bit, there are a couple of key components to making bug tracking and resolution successful on a J2EE development project, or, for that matter, any other software development project. The first component is to design a process for error resolution as part of your development methodology. The second component is to have a tool that is easy to use and provides built-in workflow and management reporting. Ideally, you would have a tracking

system that is fully integrated with your source code control system. If, for example, you use Rational Clear Case for source control, you could implement Rational Clear Quest for defect tracking. Using nonintegrated products for these functions makes the defect resolution process more manual and error-prone, limiting the usefulness of the process and hindering productivity.

On the other hand, when you are using a bare-bones approach such as CVS, the way in which problems are tracked is undefined. Problem tracking using only a source code control system is more often a manual process than an automated one, where developers might be directed to put comments into the commit logs describing the bug they have fixed. If you do not use a version control system at all, tracking modifications to your code base is as ad hoc and error-prone as all your other development activities.

1.3 *Testing and deployment in J2EE*

J2EE applications are inherently difficult to test and deploy. They are difficult to test because of the levels of indirection in the system and the nature of distributed processing itself. They are difficult to deploy because of the amount of configuration required to connect the various components such that they can collaborate. Difficulty in testing and deployment is the price we pay for the generality and flexibility of J2EE.

1.3.1 *Testing J2EE applications*

In table 1.5, we summarized the major types of testing typically done on distributed applications. Picking the types of testing your J2EE application needs is the first order of business. Often your client may dictate this information to you. Most forms of testing are usually required in one form or another, with the exception of integration testing for self-contained systems. This section describes the various types of components that require testing and suggests some strategies for doing so.

Testing thick clients

If your application does not employ a Web-based interface, you may need to separately test the thick-client side of the application. In such circumstance, you have to test the behavior of code executing in the *application client container*, the J2EE term for a JVM on a client-side machine. To make your client-side testing easier, you may choose to write simple test harnesses with predictable behavior and point your client at them instead of the J2EE server. For example, a simple test harness might be an RMI object that always returns the

same value to a caller regardless of input parameters. Using test harnesses for client components does require extra development time, but can make testing more meaningful and faster overall.

Depending on your choice of Java IDE, you may already have a debugging tool to assist you in unit testing your client-side components. For example, WebGain Studio will run your code inside its debugger, allowing you to step through executing code. This can be useful for testing components running in a local JVM. If you are not using an IDE, unit testing can still be accomplished using open source tools such as the JUnit testing framework mentioned in the previous section. There are also commercial tools on the market that provide rich functional and nonfunctional testing capabilities for applications. An example is JProbe, a Java-based testing suite from Sitraka Software. You may want to investigate these products if your IDE or open source package does not provide all of the testing metrics you require.

Testing web components

Since J2EE applications prefer the thin-client model, most J2EE test plans must accommodate some form of Web-based testing. Web components, such as servlets and JSP, must be tested over HTTP at a data (page) and protocol level. The low-tech version of this testing is performed by humans using web browsers to compare execution results to the success conditions of individual tests. The problems with this method include the amount of time and resources required, the potential for incomplete coverage of the test plan, and the possibility of human error.

Automating web unit tests can be accomplished with open source tools, including SourceForge's HTTP Unit testing framework noted earlier. Using these tools does not save much time up front, since the tests themselves must be coded. However, rerunning unit tests many times is easy, and can be an essential part of your overall code integration methodology.

For more automated and advanced web testing requirements, there are several test suites on the market that can be used in place of human testers to make web testing faster and more meaningful. In addition, these tools can perform load and performance testing as well. A popular example is the product suite offered by Mercury Interactive, which includes a functional testing tool (WinRunner) and a performance testing tool (LoadRunner). These tools do not eliminate the need for human testers, as the tests must be scripted by someone. However, once the test scripts have been recorded and the completeness of the test plan verified, running tests and collecting meaningful statistics is much easier.

Testing EJB components

Testing EJB components is the most difficult part of J2EE testing. Testing whether a behavior is executing properly is relatively simple, but determining the root cause of any errors often requires some detective work. In general, testing your EJB components requires a two-phase approach. The first occurs during development, when detailed logging is built into the EJB methods themselves. Note that, if you are using JDK version prior to 1.4, this logging capability should be encapsulated into its own subsystem (see the Façade software pattern) so that your components don't become dependent on your vendor's proprietary logging mechanisms. This is depicted in figure 1.8.

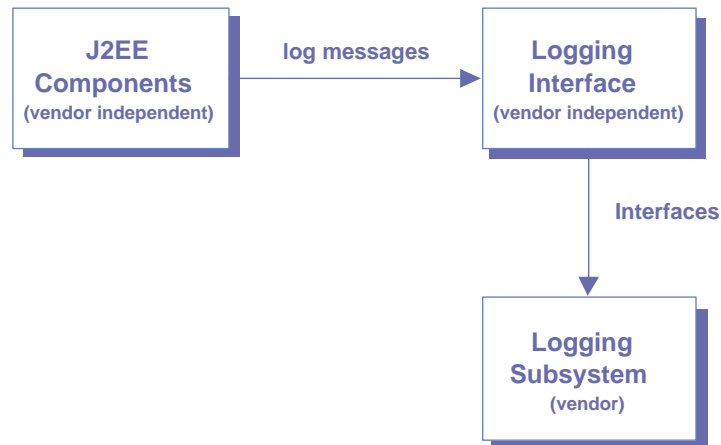


Figure 1.8 Logging adapter mechanism

Rather than creating your own logging infrastructure from scratch or using your vendor's logging API, you may decide to standardize on an open source logging API such as Log4j from the Apache Software Foundation. Information on Log4j can be found at <http://jakarta.apache.org>.

If you are already using JDK version 1.4 or later as you read this, your logging can be done via the standard Java API classes in the package `java.util.logging`. Support for JDK 1.4 in J2EE server products should minimize logging implementation issues in the future.

The second phase of EJB testing is deploying the bean and thoroughly exercising its local or remote interface against some predictable results (perhaps a prepopulated test database). Apache's Cactus framework or JUnitEE are alternatives in this area, although both require a healthy amount of configuration

and test code development. The JProbe software suite also integrates with many J2EE servers for more automated EJB testing of remote interfaces.

Testing local EJBs and dependent objects

Since EJBs accessed via a remote interface should be coarse-grained components, many rely on the functionality provided by other local EJBs or dependent objects for tasks like data persistence, remote system interactions, and service interfaces. Testing an EJB that is only available locally requires testing code to be running in the same JVM as the EJB. Fortunately, this can be accomplished using Cactus or JUnitEE in most circumstances.

Testing dependent objects directly can be challenging, but using them without directly testing them can make debugging your EJB impossible. In these cases, we recommend that you design dependent objects to be very configurable and have their owning EJB pass in configuration data from the deployment descriptor at runtime. Then implement either a JUnit test case or a main method within the dependent object that configures an instance with some hard-coded values and exercises it. The dependent object can then be tested outside of the EJB prior to testing the EJB itself. Structuring tests in this manner can increase confidence that EJB level errors are not the result of misbehaved member objects.

End-to-end testing strategy

Logically sequencing and structuring your J2EE testing activities is essential to efficient testing and debugging. Figure 1.9 suggests an overall approach to testing the various types of components you are likely to have in your J2EE application. This is a bottom-up testing strategy, in which each layer builds upon the successful completion of testing from the previous layer.

Sequencing of testing phases tends to be somewhat fluid, based on the types of testing your system requires. Most testing cycles tend to follow a sequence such as the one depicted in figure 1.10. However, it is possible to

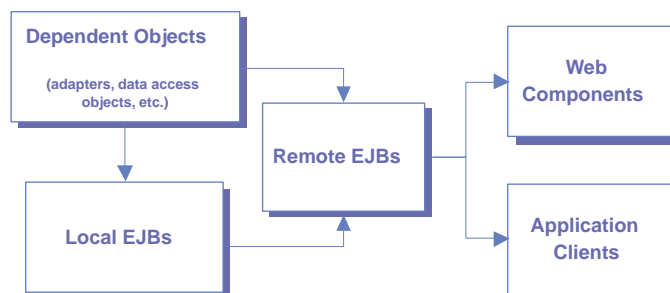


Figure 1.9
Component testing approach

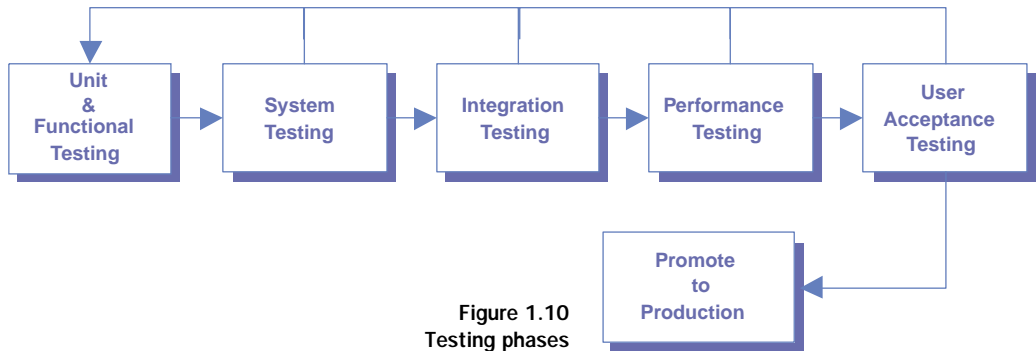


Figure 1.10
Testing phases

simultaneously test UAT and performance to reduce overall delivery time. Note that testing cycles can be iterative when necessary.

1.3.2 *Deploying J2EE applications*

J2EE's flexibility and portability can create problems for those who assemble and deploy enterprise Java applications, a situation that is complicated by the proliferation of J2EE component packaging schemes and deployment descriptor updates. In this section, we take a moment to discuss your overall deployment options and make some suggestions to help you manage your J2EE runtime configuration.

Component development and packaging

J2EE components can be deployed in various types of JAR files, as depicted in figure 1.11. When you roll your components into production, you might archive your EJB JAR files and WAR files into a single Enterprise Application Archive (EAR) file and deploy it. However, there is a large amount of vendor-specific configuration to be done for each component before it is deployed. Creating the individual components and then integrating them into an application archive is more complicated than it appears.

Current J2EE server implementations require a vendor-specific deployment descriptor file to be included with your EJB and web components. These files handle the deployment specifics that are not addressed by the generic J2EE descriptors. These specifics include nonfunctional characteristics like load balancing and failover information and resource mappings for references made in the standard deployment descriptors. In the case of EJB, you also need to run your component JAR files (including the vendor deployment descriptor) through a vendor tool to generate the specific implementation classes for your

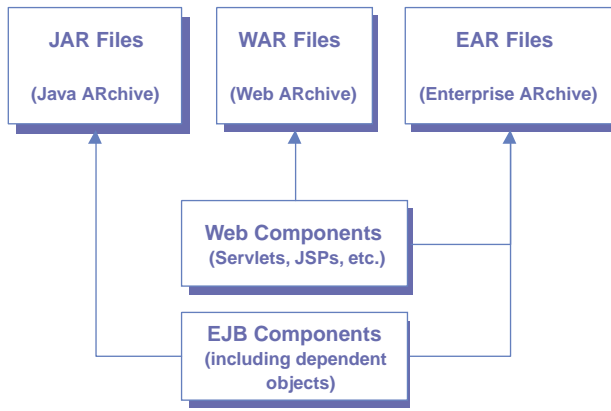


Figure 1.11
J2EE deployment formats

home and remote interfaces. During development, this process can make debugging a long and error-prone process.

To minimize the overhead of deploying your components during development, we recommend the following approach:

- Use a build tool that can be configured once to compile, jar, and deploy individual components. Ant is an excellent choice. Your chosen IDE may perform this function itself or be integrated with Ant to accomplish this.
- Deploy your web applications in expanded directory format during development. In most development tools, if you keep your web components separated into their own project, it is possible to specify that the deployment paths for your servlet classes and JSPs will be your build output directories. In this configuration, recompiling your project will update the deployed copy as well. (Note: Making changes to servlet code may require an explicit redeployment in your web container, depending on the vendor.)

In J2EE development, it is always worthwhile to spend time up front structuring and configuring your development environment, including the use of the right tool set. This will save an enormous amount of time over the life of your project and should offset the cost of any additional purchases and configuration time.

Managing component configuration

You are also likely to face issues dealing with the interdependencies among your application components at some point. For example, a given EJB might require access to a data source, a message queue, and two other remote EJBs.

These dependencies tend to grow exponentially with the complexity of the system, and managing them can become unwieldy. If you are working on a medium to large size application, consider centralizing configuration-related code using the J2EE Service Locator pattern. You can use this technique to remove complexity from individual components and centralize access to your configuration data. If you are unfamiliar with the Service Locator design pattern, refer to appendix A for more information.

An example of this strategy is the use of a JNDI Service Locator component. This component could be a local Session bean that contains all the JNDI configuration data and mappings in its deployment descriptor. Your other components can query this bean via a local interface to obtain handles to data sources, message queues, and other beans using application-wide identifiers or class references. For example, an EJB might be found by passing its class to the Service Locator. A message queue might be found by passing in a global name that the Service Locator has mapped to a JMS queue configured in its deployment descriptor. This approach can be quite useful in systems consisting of more than a handful of components, or in an environment where multiple external resources must be accessed throughout the application.

1.4 Summary

This chapter covered a lot of ground in the areas of distributed computing and J2EE development. The goal was to give you an appreciation for the challenges you will face—and the tools that will help you face them—when building a J2EE-XML system.

A distributed system is a set of independent processes that communicate with each other by passing messages over a communication channel. The client/server and peer processing models are the most common architectures in use today, and they are often combined to create more flexible distributed systems.

A distributed application relies heavily on a layered approach to software development using middleware. Middleware abstracts differences in computing environments and provides a common set of services for applications built on top of it. This overcomes the wide diversity among system components, which is the common challenge of devising distributed systems. This is the *raison d'être* of the J2EE platform, which is a vendor-independent form of middleware.

The n-tier architectural model is a common, useful tool for building various types of application components. This model dissects the application into presentation, application logic, data, and service layers for purposes of analyzing

and designing different types of functionality. We use the n-tier model to structure our detailed discussions on combining J2EE and XML in the remainder of the book. Chapter 3 discusses the application logic and data layers. Chapter 4 covers the services layer. Chapter 5 examines the presentation layer. Chapter 6 combines all the layers into a cohesive, n-tier application.

Beyond the need for middleware, common challenges in distributed development include ensuring system flexibility/extendability, vendor independence, scalability, performance, concurrency, fault masking, transparency, and security. Strategies exist to address each of these, and your J2EE vendor provides tools that implement many of those strategies.

The role of formal methodologies in your J2EE development projects depends on the size of your team, the length of your project, and the number of artifacts you need to produce during analysis and design. RUP and XP are good examples of two ends of the methodology spectrum, and we noted the conditions under which each is most applicable. More importantly, we also abstracted a few common principles from existing methodologies that can be used in the creation of your own process or the customization of an existing one.

In section 1.2, we took a brief tour of the categories of tools required in J2EE development and pointed out a few popular choices in each category. Many of these are open source and widely used, even in conjunction with commercial products. Others are popular, commercial products that either integrate well with your chosen server or provide functionality not readily available in open source tools. The goal here was to create a checklist of required tools and identify any holes in your development environment in this regard.

Section 1.3 discussed complicated issues regarding the testing and deployment of a J2EE application. We discussed useful approaches to testing various components and deploying them to your server environment, with an emphasis on build-time processes.

From here, we turn our attention to the specifics of using XML technology in the J2EE environment. Remaining chapters assume your mastery of the material in this chapter and demonstrate in detail the integration of J2EE and XML at each layer of an n-tiered architecture.



XML and Java



This chapter

- Describes relevant XML standards and technologies
- Classifies XML tools in terms of functionality
- Introduces and demonstrates use of Java XML Pack APIs (JAX)
- Suggests how JAX APIs are best deployed in your architecture

A complex set of closely related XML APIs, each of which is either in specification or development, is the result of a flurry of Java community development activity in the area of XML. These APIs include the JAX family, as well as other popular emerging standards such as JDOM.

This chapter untangles the web of Java APIs for XML, identifying and classifying each in terms of its functionality, intended use, and maturity. Where possible, we provide usage examples for each new API and describe how it might be best used in your J2EE system. We also identify areas in which the APIs overlap and suggest which ones are likely to be combined or eliminated in the future. Subsequent chapters build upon your understanding of these APIs by providing more specific examples of their implementation.

To fully appreciate the capabilities and limitations of the current JAX APIs, section 2.1 provides a brief overview of the state of important XML technologies. These technologies and standards are implemented and used by the JAX APIs, so understanding something about each will speed your mastery of JAX.

2.1 *XML and its uses*

Before diving into the details of Java's XML API family, a brief refresher on a few important XML concepts is warranted. This section provides such a refresher, as well as an overview of the most important recent developments in XML technology.

XML, the eXtensible Markup Language, is not actually a language in its own right. It is a metalanguage used to construct other languages. XML is used to create structured, self-describing documents that conform to a set of rules created for each specific language. XML provides the basis for a wide variety of industry- and discipline-specific languages. Examples include Mathematical Markup Language (MathML), Electronic Business XML (ebXML), and Voice Markup Language (VXML). This concept is illustrated in figure 2.1.

XML consists of both markup and content. *Markup*, also referred to as *tags*, describes the content represented in the document. This flexible representation of data allows you to easily send and receive data, and transform data from one format to another. The uses of XML are rapidly expanding and are partially the impetus for writing this book. For example, business partners use XML to exchange data with each other in new and easier ways. E-business related information such as pricing, inventory, and transactions are represented in XML and transferred over the Internet using open standards and protocols. There are also many specialized uses of XML, such as the Java Speech Markup Language and the Synchronized Multimedia Integration Language.

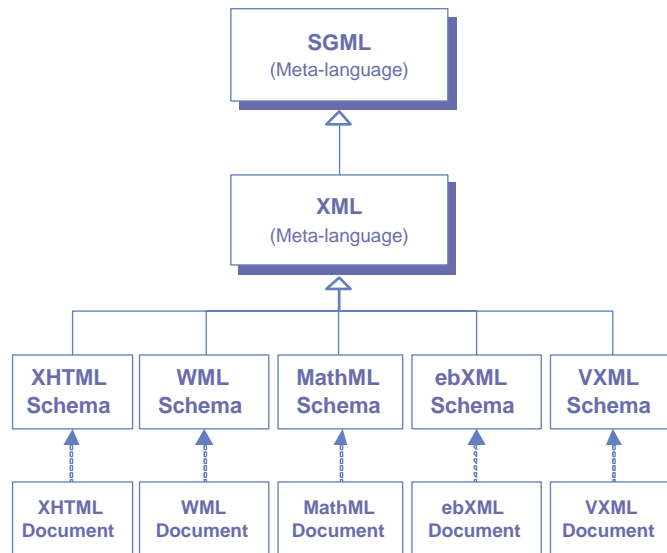


Figure 2.1
XML language
hierarchy

Each XML language defines its own *grammar*, a specific set of rules governing the content and structure of documents written in that language. For example, the element `price` may be valid in an ebXML document but has no meaning in a MathML document. Since each language must fulfill this grammatical requirement, XML provides facilities for generically documenting the correct grammar of any derived language. Any XML parser can validate the structure of any XML document, given the rules of its language.

Using XML as a common base for higher-level languages enables the interchange of data between software components, systems, and enterprises. Parsing and translation tools written to handle any type of XML-based data can be employed to create and manipulate data in a uniform way, regardless of each document's semantic meaning. For example, the same XML parser can be used to read a MathML document and an ebXML document, and the same XML Translator can be used to convert an ebXML purchase order document into a RosettaNet PIP document.

An XML-based infrastructure enables high levels of component reuse and interoperability in your distributed system. It also makes your system interfaces cleaner and more understandable to those who must maintain and extend it. And since XML is an industry standard, it can be deployed widely in your systems without worry about vendor dependence. XML also makes sense from the standpoint of systems integration, as an alternative to distributed object interaction. It allows data-level integration, making the coupling

between your application and other systems much looser and enhancing overall architectural flexibility.

In addition to its uses in messaging and data translation, XML can also be used as a native data storage format in some situations. It is particularly well suited for managing document repositories and hierarchical data. We examine some of the possibilities in this area in chapter 3.

An example XML document

To illustrate the power and flexibility of XML and related technologies, we need a concrete XML example with which to work. We use this simple document throughout the rest of this chapter to illustrate the use of various XML technologies. Most importantly, we use it to demonstrate the use of the JAX APIs in section 2.2.

Listing 2.1 contains an XML *instance document*, a data structure containing information about a specific catalog of products.

Listing 2.1 Product XML document example

```
<?xml version="1.0"?>
<product-catalog>
  <product sku="123456" name="The Product">
    <description locale="en_US">
      An excellent product.
    </description>
    <description locale="es_MX">
      Un producto excelente.
    </description>
    <price locale="en_US" unit="USD">
      99.95
    </price>
    <price locale="es_MX" unit="MXP">
      9999.95
    </price>
  </product>
</product-catalog>
```

Defines a product with SKU=123456 and the name "The Product"

1 Lists descriptions and prices for this product in the U.S. and Mexico

- 1 Shows a catalog containing a single product. The product information includes its name, SKU number, description, and price. Note that the document contains multiple price and description nodes, each of which is specific to a locale.

Classifying XML technologies

There are numerous derivative XML standards and technologies currently under development. These are not specific to Java, or any other implementation

language for that matter. They are being developed to make the use of XML easier, more standardized, and more manageable. The widespread adoption of many of them is critical to the success of XML and related standards.

This section provides a brief overview of the most promising specifications in this area. Since it is impossible to provide exhaustive tutorials for each of these in this section, we recommend you visit <http://www.zvon.org>, a web site with excellent online tutorials for many of these technologies.

2.1.1 XML validation technologies

The rules of an XML language can be captured in either of two distinct ways. When codified into either a document type definition or an XML schema definition, any validating XML parser can enforce the rules of a particular XML dialect generically. This removes a tremendous burden from your application code. In this section, we provide a brief overview of this important feature of XML.

Document type definitions

The first and earliest language definition mechanism is the document type definition (DTD).

DEFINITION A *document type definition* is a text file consisting of a set of rules about the structure and content of XML documents. It lists the valid set of elements that may appear in an XML document, including their order and attributes.

A DTD dictates the hierarchical structure of the document, which is extremely important in validating XML structures. For example, the element `Couch` may be valid within the element `LivingRoom`, but is most likely not valid within the element `BathRoom`. DTDs also define element attributes very specifically, enumerating their possible values and specifying which of them are required or optional.

Listing 2.2 DTD for the product catalog example document

<pre><!ELEMENT product-catalog (product+)> <!ELEMENT product (description+, price+)> <!ATTLIST product sku ID #REQUIRED name CDATA #REQUIRED > <!ELEMENT description (#PCDATA)></pre>	<table border="0"> <tr> <td style="border-left: 1px solid black; padding-left: 5px;"> Each product must have a SKU and name attribute </td> <td style="border-left: 1px solid black; padding-left: 5px;"> Product catalogs must contain one or more products and each product has one or more descriptions and one or more prices </td> </tr> </table>	Each product must have a SKU and name attribute	Product catalogs must contain one or more products and each product has one or more descriptions and one or more prices
Each product must have a SKU and name attribute	Product catalogs must contain one or more products and each product has one or more descriptions and one or more prices		


```
<!ATTLIST description
  locale CDATA #REQUIRED
>
<!ELEMENT price (#PCDATA)>
  <!ATTLIST price
    locale CDATA #REQUIRED
    unit CDATA #REQUIRED
  >
```

Listing 2.2 contains a DTD to constrain our product catalog example document. For this DTD to be used by a validating XML parser, we could add the DTD in-line to listing 2.1, right after the opening XML processing instruction. We could also store the DTD in a separate file and reference it like this:

```
<!DOCTYPE product-catalog SYSTEM "product-catalog.dtd">
```

Using this statement, a validating XML parser would locate a file named `product-catalog.dtd` in the same directory as the instance document and use its contents to validate the document.

XML Schema definitions

Although a nice first pass at specifying XML languages, the DTD mechanism has numerous limitations that quickly became apparent in enterprise development. One basic and major limitation is that a DTD is not itself a valid XML document. Therefore it must be handled by XML parsing tools in a special way.

More problematic, DTDs are quite limited in their ability to constrain the structure and content of XML documents. They cannot handle namespace conflicts within XML structures or describe complex relationships among documents or elements. DTDs are not modular, and constraints defined for one data element cannot be reused (inherited) by other elements. For these reasons and others, the World Wide Web Consortium (W3C) is working feverishly to replace the DTD mechanism with XML Schema.

DEFINITION An *XML Schema definition (XSD)* is an XML-based grammar declaration for XML documents.

XSD is itself an XML language. Using XSD, data constraints, hierarchical relationships, and element namespaces can be specified more completely than with DTDs. XML Schema allows very precise definition of both simple and complex data types, and allows types to inherit properties from other types.

There are numerous common data types already built into the base XML Schema language as a starting point for building specific languages. Listing 2.3 shows a possible XML Schema definition for our example product catalog document.

Listing 2.3 An XSD for the product catalog document

```

<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
  <xsd:element type="product-catalog" />
  <xsd:complexType name="productCatalog">
    <xsd:element type="productType"
      minOccurs="1" />
  </xsd:complexType>
  <xsd:complexType name="productType">
    <xsd:element name="description"
      type="xsd:string" minOccurs="1">
      <xsd:attribute name="locale"
        type="xsd:string" />
    </xsd:element>
    <xsd:element name="price"
      type="xsd:decimal" minOccurs="1">
      <xsd:attribute name="locale"
        type="xsd:string" />
      <xsd:attribute name="unit"
        type="xsd:string" />
    </xsd:element>
    <xsd:attribute name="sku"
      type="xsd:decimal" />
    <xsd:attribute name="name"
      type="xsd:string" />
  </xsd:complexType>
</xsd:schema>

```

“xsd” namespace defined by XML Schema

Declares the product catalog

Defines catalog type containing one or more product elements

1 product type definition

- 1 This XSD defines a complex type called `productType`, which is built upon other primitive data types. The complex type contains attributes and other elements as part of its definition. Just from the simple example, the advantages of using XML Schema over DTDs should be quite apparent to you.

The example XSD in listing 2.3 barely scratches the surface of the intricate structures that you can define using XML Schema. Though we will not focus on validation throughout this book, we strongly encourage you to become proficient at defining schemas. You will need to use them frequently as the use

of XML data in your applications increases. Detailed information on XML Schema can be found at <http://www.w3c.org/XML/Schema>.

Before leaving the topic of document validation, we note that some parsers do not offer any validation at all, and others only support the DTD approach. Document validation is invaluable during development and testing, but is often turned off in production to enhance system performance. Using validation is also critical when sharing data between enterprises, to ensure both parties are sending and receiving data in a valid format.

2.1.2 *XML parsing technologies*

Before a document can be validated and used, it must be parsed by XML-aware software. Numerous XML parsers have been developed, including Crimson and Xerces, both from the Apache Software Foundation. You can learn about these parsers at <http://xml.apache.org>. Both tools are open source and widely used in the industry. Many commercial XML parsers are also available from companies like Oracle and IBM.

DEFINITION An *XML parser* is a software component that can read and (in most cases) validate any XML document. A parser makes data contained in an XML data structure available to the application that needs to use it.

SAX

Most XML parsers can be used in either of two distinct modes, based on the requirements of your application. The first mode is an event-based model called the Simple API for XML (SAX). Using SAX, the parser reads in the XML data source and makes callbacks to its client application whenever it encounters a distinct section of the XML document. For example, a SAX event is fired whenever the end of an XML element has been encountered. The event includes the name of the element that has just ended.

To use SAX, you implement an event handler for the parser to use while parsing an XML document. This event handler is most often a state machine that aggregates data as it is being parsed and handles subdocument data sets independently of one another. The use of SAX is depicted in figure 2.2. SAX is the fastest parsing method for XML, and is appropriate for handling large documents that could not be read into memory all at once.

One of the drawbacks to using SAX is the inability to look forward in the document during parsing. Your SAX handler is a state machine that can only

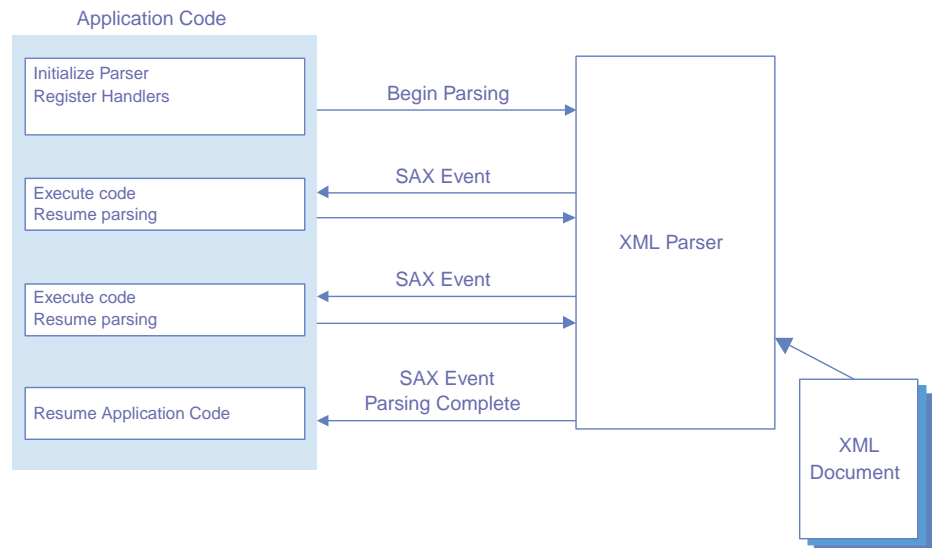


Figure 2.2 Using the SAX API

operate on the portion of the XML document that has already been parsed. Another disadvantage is the lack of predefined relationships between nodes in the document. In order to perform any logic based on the parent or sibling nodes, you must write your own code to track these relationships.

DOM

The other mode of XML parsing is to use the Document Object Model (DOM) instead of SAX. In the DOM model, the parser will read in an entire XML data source and construct a treelike representation of it in memory. Under DOM, a pointer to the entire document is returned to the calling application. The application can then manipulate the document, rearranging nodes, adding and deleting content as needed. The use of DOM is depicted in figure 2.3.

While DOM is generally easier to implement, it is far slower and more resource intensive than SAX. DOM can be used effectively with smaller XML data structures in situations when speed is not of paramount importance to the application. There are some DOM-derivative technologies that permit the use of DOM with large XML documents, which we discuss further in chapter 3.

As you will see in section 2.2, the JAXP API enables the use of either DOM or SAX for parsing XML documents in a parser-independent manner. Deciding which method to use depends on your application's requirements for speed, data manipulation, and the size of the documents upon which it operates.

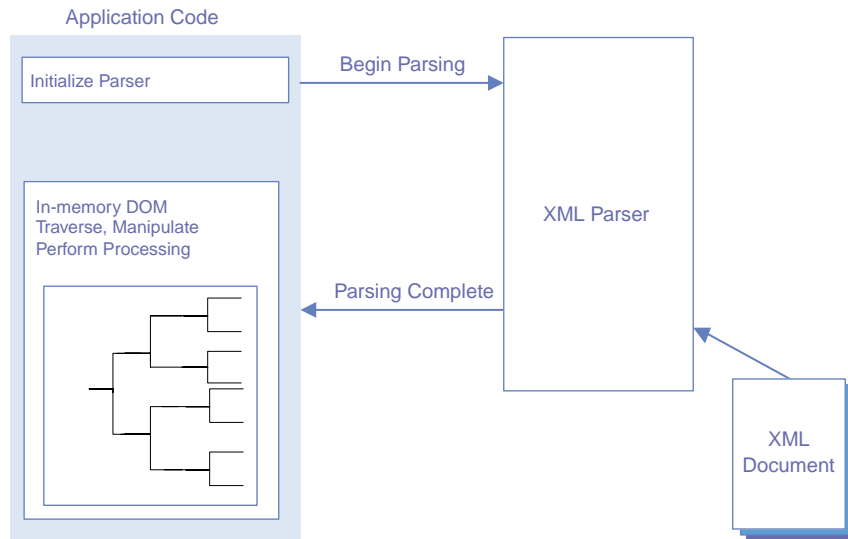


Figure 2.3 Using the DOM API

2.1.3 XML translation technologies

A key advantage of XML over other data formats is the ability to convert an XML data set from one form to another in a generic manner. The technology that enables this translation is the eXtensible Stylesheet Language for Transformations (XSLT).

XSLT

Simply stated, XSLT provides a framework for transforming the structure of an XML document. XSLT combines an input XML document with an XSL stylesheet to produce an output document.

DEFINITION An *XSL stylesheet* is a set of transformation instructions for converting a *source* XML document to a *target* output document.

Figure 2.4 illustrates the XSLT process. Performing XSLT transformations requires an XSLT-compliant processor. The most popular open source XSLT engine for Java is the Apache Software Foundation's Xalan project. Information about Xalan can be found at <http://xml.apache.org/xalan-j>.

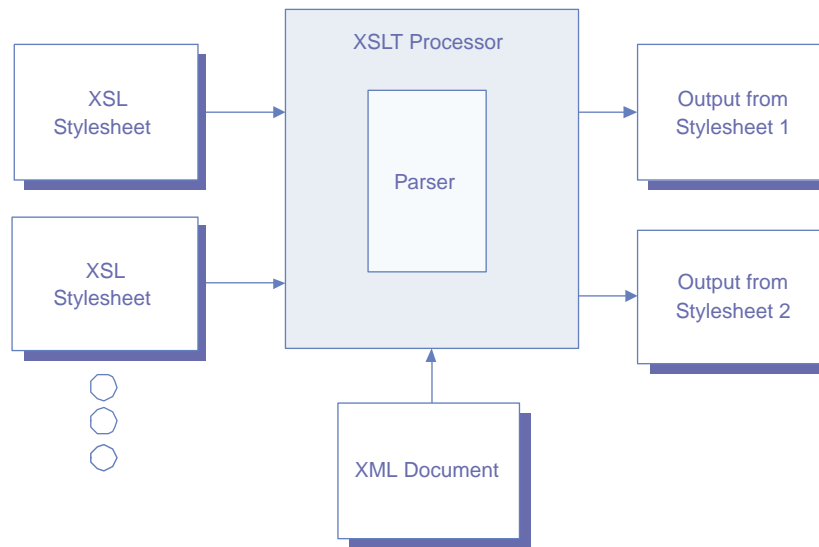


Figure 2.4 XSLT processing overview

An XSLT processor transforms an XML source tree by associating patterns within the source document with XSL stylesheet templates that are to be applied to them. For example, consider the need to transform our product catalog XML document into HTML for rendering purposes. This consists of wrapping the appropriate product data in the XML document with HTML markup. Listing 2.4 shows an XSL stylesheet that would accomplish this task.

Listing 2.4 Translating the product catalog for the Web

```

<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:template match="/">
<html>
  <head><title>My Products</title></head>
  <body>
    <h1>Products Currently For Sale in the U.S.</h1>
    <xsl:for-each select="//product">
      <xsl:value-of select="@name"/> : $
      <xsl:value-of select="./price[@unit='USD']"/> USD
    </xsl:for-each>
  
```

❶ Executes for the root element of the source document

❷ Prints name and price information

```
</body>  
</html>  
</xsl:template>  
</xsl:stylesheet>
```

- ❶ The `match` attribute is an XPath expression meaning the root XML element. This template is therefore executed against the entire source document.
- ❷ Each product element in the source document will have its name attribute printed, followed by the string: \$, its price in dollars, and the string USD.

XSLT processors can vary in terms of their performance characteristics. Most offer some way to precompile XSL stylesheets to reduce transformation times. As you will see in section 2.2, the JAXP API provides a layer of pluggability for compliant XSLT processors in a manner similar to parsers. This permits the replacement of one XSLT engine with another, faster one as soon as it becomes available.

Details on XSLT can be found at <http://www.w3.org/Style/XSL>.

Binary transformations for XML

Note that the capabilities of XSLT are not limited to textual transformations. It is often necessary to translate textual data to binary format. A common example is the translation of business data to PDF format for display. For this reason the XSL 1.0 Recommendation also specifies a set of *formatting objects*. Formatting objects are instructions that define the layout and presentation of information. Formatting objects are most useful for print media and design work. Some Java libraries are already available to do the most common types of transformations. See chapter 5 for an example of the most common binary transformation required today, from XML format to PDF.

2.1.4 *Messaging technologies*

Numerous technologies for transmitting XML-structured data between applications and enterprises are currently under development. This is due to the tremendous potential of XML to bridge the gap between proprietary data formats and messaging protocols. Using XML, companies can develop standard interfaces to their systems and services to which present and future business partners can connect with little development effort. In this section, we provide a brief description of the most promising of these technologies.

SOAP

By far the most promising advances in this area are technologies surrounding the Simple Object Access Protocol (SOAP).

DEFINITION *SOAP* is a messaging specification describing data encoding and packaging rules for XML-based communication.

The SOAP specification describes how XML messages can be created, packaged, and transmitted between systems. It includes a binding (mapping) for the HTTP protocol, meaning that SOAP messages can be transmitted over existing Web systems. Much of SOAP is based upon *XML-RPC*, a specification describing how remote procedure calls can be executed using XML.

SOAP can be implemented in a synchronous (client/server) or asynchronous fashion. The synchronous method (RPC-style) involves a client explicitly requesting some XML data from a SOAP server by sending a SOAP request message. The server returns the requested data to the client in a SOAP response message. This is depicted in figure 2.5.

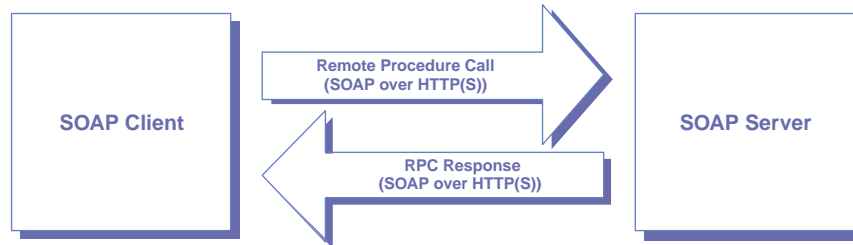


Figure 2.5 RPC-style SOAP messaging

Asynchronous messaging is also fully supported by the SOAP specification. This can be useful in situations where updates to information can be sent and received as they happen. The update event must not require an immediate response, but an asynchronous response might be sent at some point in the future. This response might acknowledge the receipt of the original message and report the status of processing on the receiver side. Asynchronous SOAP is depicted in figure 2.6.

Many J2EE server vendors now support some form of SOAP messaging, via their support of the JAXM API discussed later in this chapter. More information on the SOAP specification is available at <http://www.w3c.org/TR/SOAP>.

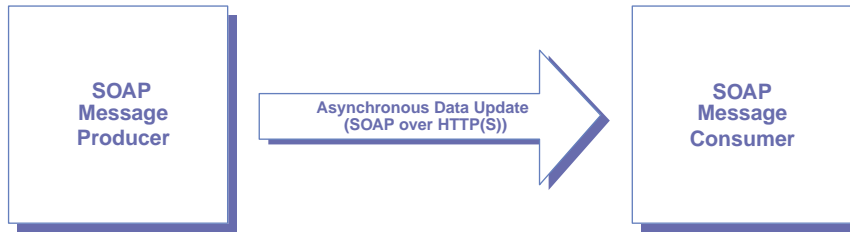


Figure 2.6 Message-style SOAP messaging

Web services

Closely related to the development of SOAP is the concept of *web services*. As we alluded to in chapter 1, web services is the catchall phrase for the standardization of distributed business services architecture over the Internet. Web services rely on SOAP clients and servers to transport inter-enterprise messages.

The subjects of XML messaging and web services are quite complex. We take a detailed look at these topics in chapter 4, including examples. In this section, we discuss only the basics of web services and related technologies.

Work is also ongoing to define a standard way to register and locate new web services using distributed service repositories, or search engines. These repositories use XML to describe web services and the companies that provide them. The most promising of these standards to date is the Universal Description, Discovery, and Integration (UDDI) specification. This is due to the broad vendor support UDDI currently enjoys from many companies, including IBM and Microsoft.

UDDI

A consortium of large companies has come together to create a set of standards around the registration and discovery process for web services. The result is UDDI. The goal of UDDI is to enable the online registration and lookup of web services via a publicly available repository, similar in operation to the Domain Name System (DNS) of the Internet. The service registry is referred to as the *green pages* and is defined in an XML Schema. The green pages are syndicated across multiple *operator sites*. Each site provides some level of public information regarding the services. This information is represented as metadata and known as a *tModel*.

One of the challenges when registering a web service is deciding on how it should be classified. A mere alphabetical listing by provider would make it impossible to find a particular type of service. UDDI therefore allows classification of

services by geographic region and standard industry codes, such as NAICS and UN/SPC. Many expect the other services repositories, such as the ebXML Repository, to merge with UDDI in the future, although no one can say for sure.

You can read more about UDDI and related technologies at <http://www.uddi.org>.

WSDL

The creators of the UDDI directory recognized the need for a standard means for describing web services in the registry. To address this, they created the Web Services Description Language (WSDL). WSDL is an XML language used to generically describe web services. The information contained in each description includes a network address, protocol, and a supported set of operations. We will discuss WSDL in detail and provide examples of it in chapter 4.

2.1.5 Data manipulation and retrieval technologies

Storing and retrieving data in XML format is the subject of much ongoing work with XML. The need for XML storage and retrieval technologies has resulted in the creation of a large number of closely related specifications. In this section, we provide you with a brief overview of these specifications and point you in the direction of more information about each.

XPath

XPath is a language for addressing XML structures that is used by a variety of other XML standards, including XSLT, XPointer, and XQuery. It defines the syntax for creating *expressions*, which are evaluated against an XML document. For example, a forward slash (/) is a simple XPath expression. As you saw in listing 2.4, this expression represents the root node of an XML document.

XPath expressions can represent a node-set, Boolean, number, or string. They can start from the root element or be relative to a specific position in a document. The most common type of XPath expression is a location path, which represents a node-set. For our product catalog document example, the following XPath represents all the product nodes in the catalog:

```
/product
```

XPath has a built-in set of functions that enable you to develop very complex expressions. Although XPath syntax is not a focus of this book, we do explore technologies such as XSLT that use it extensively. Since XPath is so important, we suggest that you become proficient with it as quickly as possible.

You can get more detailed information on XPath at <http://www.w3c.org/TR/xpath>.

XPointer

XPointer is an even more specific language that builds on XPath. XPointer expressions point to not only a node-set, but to the specific position or range of positions within a node-set that satisfy a particular condition. XPointer functions provide a very robust method for searching through XML data structures. Take, for example, the following node-set:

```
<desc>This chapter provides an overview of the J2EE technologies.</desc>  
<desc>This chapter provides an overview of the XML landscape.</desc>  
<desc>This chapter is an introduction to distributed systems.</desc>
```

A simple XPointer expression that operates on this node-set is as follows:

```
xpointer(string-range(//desc, 'overview'))
```

This expression returns all nodes with the name `desc` that contain the string `overview`. XPointer expressions can be formed in several ways and can quickly become complex. You can find more information on XPointer at <http://www.w3c.org/XML/Linking>.

XInclude

XInclude is a mechanism for including XML documents inside other XML documents. This allows us to set up complex relationships among multiple XML documents. It is accomplished by using the `<include>` tag, specifying a location for the document, and indicating whether or not it should be parsed. The `include` tag may be placed anywhere within an XML document. The location may reference a full XML document or may use XPointer notation to reference specific portions of it. The use of XPointer with XInclude makes it easier to include specific XML data and prevents us from having to duplicate data in multiple files.

Adding the following line to an XML document would include a node-set from an external file called `afile.xml` in the current XML document, at the current location:

```
<xi:include href="afile.xml#xpointer( XPath expression )" parse="xml" />
```

Only the nodes matching the specified XPath expression would be included.

More information on XInclude can be found at <http://www.w3c.org/TR/xinclude>.

XLink

XLink is a technology that facilitates linking resources within separate XML documents. It was created because requirements for linking XML resources require a more robust mechanism than HTML-style hyperlinks can provide. HTML hyperlinks are unidirectional, whereas XLink enables traversal in both directions. XLinks can be either simple or extended. Simple XLinks conform to similar rules as HTML hyperlinks, while extended XLinks feature additional functionality.

The flexibility of XLink enables the creation of extremely complex and robust relationships. The following example uses a simple XLink that establishes a relationship between an order and the customer who placed it.

If this XML document represents a customer:

```
<customer id="0059">
  <name>ABC Company</name>
  <employees>1000-1500</employees>
</customer>
```

This XML document lists orders linked to that customer:

```
<orders>
  <order xlink:type="simple"
    href="customers.xml#/customers/customer/@id[.='0059']"
    title="Customer" show="new">
    <number>12345</number>
    <amount>$500</amount>
  </order>
</orders>
```

Note once again the importance of XPath expressions in enabling this technology. More information on XLink is at <http://www.w3c.org/XML/Linking>.

XBase

XBase, or XML Base, is a mechanism for specifying a base uniform resource identifier (URI) for XML documents, such that all subsequent references are inferred to be relative to that URI. Despite its simplicity, XBase is extremely handy and allows you to keep individual XLinks to a reasonable length.

The following line describes a base URI using XBase. Any relative URI reference encountered inside the `catalog` element will be resolved using <http://www.manning.com/books> as its base.

```
<catalog xml:base=http://www.manning.com/books/>
  .....
  .....
</catalog>
```

You can learn more about XBase at <http://www.w3c.org/TR/xmlbase>.

Query languages

As the amount of data being stored in XML has increased, it is not surprising that several query languages have been developed specifically for XML. One of the initial efforts in this area was XQL, the XML Query Language. XQL is a language for querying XML data structures and shares many of its constructs with XPath. Using XQL, queries return a set of nodes from one or more documents. Other query languages include Quilt and XML-QL.

The W3C has recently taken on the daunting task of unifying these specifications under one, standardized query language. The result of this effort is a language called XQuery. It uses and builds upon XPath syntax. The result of an XML query is either a node-set or a set of primitive values. XQuery is syntactically similar to SQL, with a set of keywords including FOR, LET, WHERE, and RETURN.

The following is a simple XQuery expression that selects all product nodes from `afile.xml`.

```
document( "afile.xml" )//product
```

A slightly more complex XQuery expression selects the `warranty` node for each `product`.

```
FOR $product in //product  
  RETURN $product/warranty
```

XQuery is in its early stages of completion and there are not many products around that fully implement the specification. The latest version of Software AG's Tamino server has some support for XQuery, but a full XQuery engine has yet to be implemented. We discuss XQuery in more detail in chapter 3, within our discussion of XML data persistence. You can get all the details about XQuery at <http://www.w3c.org/XML/Query>.

2.1.6 *Data storage technologies*

XML is data, so it should be no surprise that there are a variety of technologies under development for storing native XML data. The range of technologies and products is actually quite large, and it is still unclear which products will emerge as the leaders.

Storing XML on the file system is still very popular, but storing XML in a textual, unparsed format is inefficient and greatly limits its usability. Static documents require reparsing each time they are accessed. An alternative mechanism to storing text files is the Persistent Document Object Model (PDOM). PDOM implements the W3C DOM specification but stores the parsed XML

document in binary format on the file system. In this fashion, it does not need to be reparsed for subsequent access.

PDOM documents may be generated from an existing DOM or through an XML input stream, so the document is not required to be in memory in its entirety at any given time. This is advantageous when dealing with large XML documents. PDOM supports all of the standard operations that you would expect from a data storage component, such as querying (via XQL), inserting, deleting, compressing, and caching. We offer an example of using this technique for data storage in chapter 3. You can learn more about PDOM at <http://xml.darmstadt.gmd.de/xql/>.

Another alternative to static file system storage is the use of native-XML databases. Databases such as Software AG's Tamino are designed specifically for XML. Unlike relational databases, which store hierarchical XML documents in relational tables, Tamino stores XML in its native format. This gives Tamino a significant performance boost when dealing with XML.

Despite the appearance of native-XML database vendors, traditional database vendors such as Oracle and IBM had no intention of yielding any of the data storage market just because traditional relational databases did not handle XML well initially. The major relational vendors have built extensions for their existing products to accommodate XML as a data type and enable querying functionality. This is advantageous for many companies that rely heavily on RDBMS products and have built up strong skill-sets in those technologies.

Figure 2.7 summarizes your options for XML data storage.

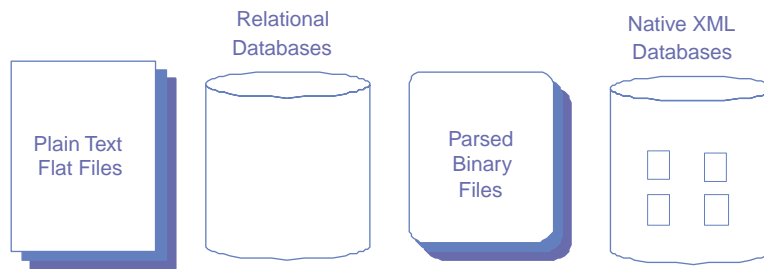


Figure 2.7 XML data storage alternatives

2.2 *The Java APIs for XML*

The Java development community is actively following and contributing to the specification of many of the XML technologies discussed in section 2.1.

To buy this book

Additionally, Java is often the first language to implement these emerging technologies. This is due largely to the complimentary nature of platform independent code (Java) and data (XML). However, XML API development in Java has historically been disjointed, parallel, and overly complicated. Various groups have implemented XML functionality in Java in different ways and at different times, which led to the proliferation of overlapping, noncompatible APIs.

To address this issue and make developing XML-aware applications in Java simpler, Sun Microsystems is now coordinating Java XML API development via the Java Community Process (JCP). Under this process, the Java development community is standardizing and simplifying the various Java APIs for XML. Most of these efforts have been successful, although a couple of the standard specifications still have overlapping scope or functionality. Nevertheless, XML processing in Java has come a long way in 2000 and 2001.

The Java APIs for XML (JAX) is currently a family of related API specifications. The members of the JAX family are summarized in table 2.1. In this section, we introduce each member of JAX and discuss its current state of maturity. For those JAX members with an existing reference implementation, we also provide usage examples for each.

Table 2.1 The JAX family—Java APIs for XML processing

Java API for XML	JAX acronym	Functional description
Java API for XML parsing	JAXP	Provides implementation-neutral access to XML parsers and XSLT processors.
Java Document Object Model	JDOM	Provides a Java-centric, object-oriented implementation of the DOM framework.
Java API for XML binding	JAXB	Provides a persistent XML mapping for Java object storage as XML.
Long Term Java-Beans Persistence		Similar to JAXB, provides XML serialization for JavaBean components.
Java API for XML messaging	JAXM	Enables the use of SOAP messaging in Java applications, using resource factories in a manner similar to the Java Messaging Service (JMS).
JAX-RPC	JAX-RPC	An XML-RPC implementation API for Java. Similar to JAXM.
Java API for XML repositories	JAXR	Provides implementation-neutral access to XML repositories like ebXML and UDDI.

2.2.1 JAXP

JAXP provides a common interface for creating and using the SAX, DOM, and XSLT APIs in Java. It is implementation- and vendor-neutral. Your applications should use JAXP instead of accessing the underlying APIs directly to enable the replacement of one vendor's implementation with another as desired. As faster or better implementations of the base XML APIs become available, you can upgrade to them simply by exchanging one JAR file for another. This achieves a primary goal in distributed application development: flexibility.

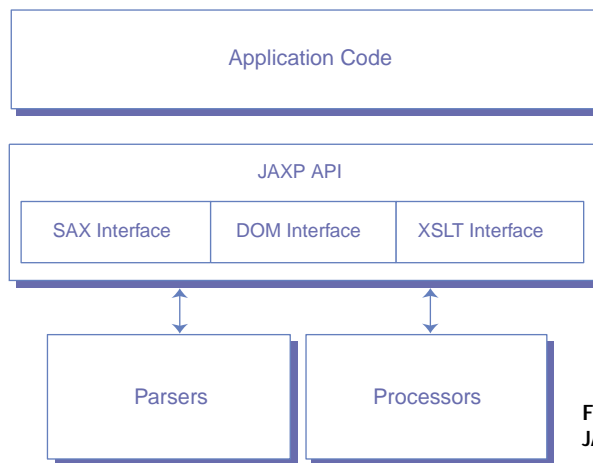


Figure 2.8
JAXP architecture

The JAXP API architecture is depicted in figure 2.8. JAXP enables flexibility by divorcing your application code from the underlying XML APIs. You can use it to parse XML documents using SAX or DOM as the underlying strategy. You can also use it to transform XML via XSLT in a vendor-neutral way.

Table 2.2 The JAXP packages

Package	Description
javax.xml.parsers	Provides a common interface to DOM and SAX parsers.
javax.xml.transform	Provides a common interface to XSLT processors.
org.xml.sax	The generic SAX API for Java
org.w3c.dom	The generic DOM API for Java

The JAXP API consists of four packages, summarized in table 2.2. Of these, the two `javax.xml` packages are of primary interest. The `javax.xml.parsers` package contains the classes and interfaces needed to parse XML documents. The `javax.xml.transform` package defines the interface for XSLT processing.

Configuring JAXP

To use JAXP for parsing, you require a JAXP-compliant XML parser. The JAXP reference implementation uses the Crimson parser mentioned earlier. To do XSLT processing, you also need a compliant XSLT engine. The reference implementation uses Xalan, also mentioned earlier.

When you first access the JAXP parsing classes in your code, the framework initializes itself by taking the following steps:

- It initially checks to see if the system property `javax.xml.parsers.DocumentBuilderFactory` OR `javax.xml.parsers.SAXParserFactory` has been set (depending on whether you are requesting the use of SAX or DOM). If you are requesting an XSLT transformation, the system property `javax.xml.transform.TransformerFactory` is checked instead.
- If the appropriate system property has not been set explicitly, the framework searches for a file called `jaxp.properties` in the `lib` directory of your JRE. Listing 2.5 shows how the contents of this file might appear.
- If the `jaxp.properties` file is not found, the framework looks for files on the classpath named `/META-INF/services/java.xml.parsers.DocumentBuilderFactory`, `/META-INF/services/SAXParserFactory`, and `/META-INF/services/javax.xml.transform.TransformerFactory`. When found, these files contain the names of the JAXP `DocumentBuilder`, `SAXParserFactory`, and `TransformerFactory` classes, respectively. JAXP-compliant parsers and XSLT processors contain these text files in their jars.
- If a suitable implementation class name cannot be found using the above steps, the platform default is used. Crimson will be invoked for parsing and Xalan for XSLT.

NOTE Statements in the following listing are shown on multiple lines for clarity. In an actual `jaxp.properties` file, each statement should appear as a single line with no spaces between the equals character (=) and the implementation class name.

Listing 2.5 A Sample jaxp.properties file

```
javax.xml.parsers.DocumentBuilderFactory=  
    org.apache.crimson.jaxp.DocumentBuilderFactoryImpl  
javax.xml.parsers.SAXParserFactory=  
    org.apache.crimson.jaxp.SAXParserFactoryImpl  
javax.xml.transform.TransformerFactory=  
    org.apache.xalan.processor.TransformerFactoryImpl
```

Sets DOM builder, SAX parser, and XSLT processor implementation classes

Since JAXP-compliant parsers and processors already contain the necessary text files to map their implementation classes to the JAXP framework, the easiest way to configure JAXP is to simply place the desired parser and/or processor implementation's JAR file on your classpath, along with the JAXP jar. If, however, you find yourself with two JAXP-compliant APIs on your classpath for some other reason, you should explicitly set the implementation class(es) before using JAXP. Since you would not want to do this in your application code, the properties file approach is probably best.

JAXP is now a part of the J2EE specification, meaning that your J2EE vendor is required to support it. This makes using JAXP an even easier choice over directly using a specific DOM, SAX, or XSLT implementation.

Using JAXP with SAX

The key JAXP classes for use with SAX are listed in table 2.3. Before demonstrating the use of SAX via JAXP, we must digress for a moment on the low level details of SAX parsing. To use SAX with or without JAXP, you must always define one or more event handlers for the parser to use.

DEFINITION A *SAX event handler* is a component that registers itself for callbacks from the parser when SAX events are fired.

The SAX API defines four core event handlers, encapsulated within the `EntityResolver`, `DTDHandler`, `ContentHandler`, and `ErrorHandler` interfaces of the `org.xml.sax` package. The `ContentHandler` is the primary interface that most applications need to implement. It contains callback methods for the `startDocument`, `startElement`, `endElement`, and `endDocument` events. Your application must implement the necessary SAX event interface(s) to define your specific implementation of the event handlers with which you are interested.

Table 2.3 Primary JAXP interfaces to the SAX API

JAXP class or interface	Description
<code>javax.xml.parsers.SAXParserFactory</code>	Locates a <code>SAXParserFactory</code> implementation class and instantiates it. The implementation class in turn provides <code>SAXParser</code> implementations for use by your application code.
<code>javax.xml.parsers.SAXParser</code>	Interface to the underlying SAX parser.
<code>javax.xml.parsers.SAXReader</code>	A class wrapped by the <code>SAXParser</code> that interacts with your SAX event handler(s). It can be obtained from the <code>SAXParser</code> and configured before parsing when necessary.
<code>org.xml.sax.helpers.DefaultHandler</code>	A utility class that implements all the SAX event handler interfaces. You can subclass this class to get easy access to all possible SAX events and then override the specific methods in which you have interest.

The other types of event handlers defined in SAX exist to deal with more peripheral tasks in XML parsing. The `EntityResolver` interface enables the mapping of references to external sources such as databases or URLs. The `ErrorHandler` interface is implemented to handle special processing of `SAXExceptions`. Finally, the `DTDHandler` interface is used to capture information about document validation as specified in the document's DTD.

SAX also provides a convenience class called the `org.xml.sax.helpers.DefaultHandler`, which implements all of the event handler interfaces. By extending the `DefaultHandler` class, your component has access to all of the available SAX events.

Now that we understand how SAX works, it is time to put JAXP to work with it. For an example, let us read in our earlier product catalog XML document using SAX events and JAXP. To keep our example short and relevant, we define a SAX event handler class that listens only for the `endElement` event. Each time a product element has been completely read by the SAX parser, we print a message indicating such. The code for this handler is shown in listing 2.6.

Listing 2.6 SAX event handler for product nodes

```
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class ProductEventHandler
    extends DefaultHandler { ← Extends this class to only handle
                               the endElement event
```

```
// other event handlers could go here
public void endElement( String namespaceURI,
                      String localName,
                      String qName,
                      Attributes atts )
    throws SAXException {
    // make sure it was a product node
    if (localName.equals("product"))
        System.out.println(
            "A product was read from the catalog.");
}
}
```

Now that we have defined an event handler, we can obtain a SAX parser implementation via JAXP in our application code and pass the handler to it. The handler's `endElement` method will be called once when parsing the example document, since there is only one product node. The code for our JAXP SAX example is given in listing 2.7.

Listing 2.7 Parsing XML with JAXP and SAX

```
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
import java.io.File;

public class JAXPandSAX {

    public static void main(String[] args) {

        ProductEventHandler handler
            = new ProductEventHandler(); | Instantiates our
                                     | event handler

        try {
            SAXParserFactory factory
                = SAXParserFactory.newInstance();
            SAXParser parser
                = factory.newSAXParser(); ← Obtains a SAXParser via JAXP
            File ourExample
                = new File("product-catalog.xml");

            parser.parse( ourExample, handler);

        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

When the code in listing 2.6 is executed against our product catalog document from listing 2.1, you should see the following output:

```
Product read from the catalog.
```

This statement only prints once, since we have only defined a single product. If there were multiple products defined, this statement would have printed once per product.

Using JAXP with DOM

Using JAXP with DOM is a far less complicated endeavor than with SAX. This is because you do not need to develop an event handler and pass it to the parser. Using DOM, the entire XML document is read into memory and represented as a tree. This allows you to manipulate the entire document at once, and does not require any state-machine logic programming on your part. This convenience comes, of course, at the expense of system resources and speed. The central JAXP classes for working with DOM are summarized in table 2.4.

Table 2.4 Primary JAXP interfaces to the DOM API

JAXP class or interface	Description
<code>javax.xml.parsers.DocumentBuilderFactory</code>	Locates a <code>DocumentBuilderFactory</code> implementation class and instantiates it. The implementation class in turn provides <code>DocumentBuilder</code> implementations.
<code>javax.xml.parsers.DocumentBuilder</code>	Interface to the underlying DOM builder.

Since our product catalog document is very short, there is no danger in reading it in via DOM. The code to do so is given in listing 2.8. You can see that the general steps of obtaining a parser from JAXP and invoking it on a document are the same. The primary difference is the absence of the SAX event handler. Note also that the parser returns a pointer to the DOM in memory after parsing. Using the other DOM API classes in the `org.w3c.dom` package, you could traverse the DOM in your code and visit each product in the catalog. We leave that as an exercise for the reader.

Listing 2.8 Building a DOM with JAXP

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import java.io.File;

public class JAXPandDOM {
```

↳ Imports the JAXP DOM classes

```

public static void main(String[] args) {
    try {
        DocumentBuilderFactory factory
            = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder
            = factory.newDocumentBuilder();
        File ourExample
            = new File("product-catalog.xml");
        Document document
            = builder.parse(ourExample);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
}

```

Obtains a DOMBuilder via JAXP

Parses the XML and builds a DOM tree

Using JAXP with XSLT

JAXP supports XSLT in the same implementation-independent manner as XML parsing. The JAXP interfaces to XSLT are located in the `javax.xml.transform` package. The primary classes and interfaces are summarized in table 2.5. In addition to these top-level interfaces, JAXP includes three subpackages to support the use of SAX, DOM, and I/O streams with XSLT. These packages are summarized in table 2.6.

Table 2.5 Primary JAXP interfaces to the XSLT API

JAXP class or interface	Description
<code>javax.xml.transform.TransformerFactory</code>	Locates a <code>TransformerFactory</code> implementation class and instantiates it.
<code>javax.xml.transform.Transformer</code>	Interface to the underlying XSLT processor.
<code>javax.xml.transform.Source</code>	An interface representing an XML data source to be transformed by the <code>Transformer</code> .
<code>javax.xml.transform.Result</code>	An interface to the output of the <code>Transformer</code> after XSLT processing.

In section 2.1.3, we discussed the XSLT process and saw how our product catalog document could be transformed into HTML via XSLT. Now we examine how that XSLT process can be invoked from your Java code via JAXP. For the sake of clarity and simplicity, we will use the I/O stream helper classes

from the `javax.xml.transform.stream` package to create our `Source` and `Result` objects.

Table 2.6 JAXP helper packages for XSLT

Package name	Description
<code>javax.xml.transform.dom</code>	Contains classes and interfaces for using XSLT with DOM input sources and results.
<code>javax.xml.transform.sax</code>	Contains classes and interfaces for using XSLT with SAX input sources and results.
<code>javax.xml.transform.stream</code>	Contains classes and interfaces for using XSLT with I/O input and output stream sources and results.

The code we need to convert our example document to HTML is shown in listing 2.9. To compile it, you must have the JAXP jar file in your classpath. To run this program, you must have the example product catalog XML document from listing 2.1 saved in a file called `product-catalog.xml`. The stylesheet from listing 2.4 must be saved to a file named `product-catalog-to-html.xsl`. You can either type these files into your favorite editor or download them from the book's web site at <http://www.manning.com/gabrick>. You will also need to place a JAXP-compliant XSLT engine (such as Xalan) in your classpath before testing this example.

Listing 2.9 Building a DOM with JAXP

```
import javax.xml.transform.*;           Imports the JAXP
import javax.xml.transform.stream.*;    XSLT API
import java.io.File;

public class JAXPandXSLT {

    public static void main(String[] args) {

        File sourceFile
            = new File("product-catalog.xml");
        File xsltFile
            = new File("product-catalog-to-html.xsl");

        Source xmlSource = new StreamSource(sourceFile);
        Source xsltSource = new StreamSource(xsltFile);
        Result result = new StreamResult(System.out);

        TransformerFactory factory
            = TransformerFactory.newInstance();

        try {
```

```
Transformer transformer
    = factory.newTransformer(xsltSource);
transformer.transform(xmlSource, result);
} catch (TransformerConfigurationException tce) {
    System.out.println("No JAXP-compliant XSLT processor found.");
} catch (TransformerException te) {
    System.out.println("Error while transforming document:");
    te.printStackTrace();
}
}
```

① Factory returns new Transformer

② Performs transformation

- ① The `TransformerFactory` implementation then provides its own specific `Transformer` implementation. Note that the transformation rules contained in the XSLT stylesheet are passed to the factory for it to create a `Transformer` object.
- ② This is the call that actually performs the XSLT transformation. Results are streamed to the specified `Result` stream, which is the console in this example.

At first glance, using XSLT via JAXP does not appear to be too complex. This is true for simple transformations, but there are many attributes of the XSLT process that can be configured via the `Transformer` and `TransformerFactory` interfaces. You can also create and register a custom error handler to deal with unexpected events during transformation. See the JAXP documentation for a complete listing of the possibilities. In this book, we concentrate on where and how you would use JAXP in your J2EE code rather than exhaustively exercising this API.

A word of caution

Using XSLT, even via JAXP, is not without its challenges. The biggest barrier to the widespread use of XSLT is currently performance. Performing an XSLT transformation on an XML document is time- and resource-intensive. Some XSLT processors (including Xalan) allow you to precompile the transformation rules contained in your stylesheets to speed throughput. Through the JAXP 1.1 interface, it is not yet possible to access this feature.

Proceed with caution and perform thorough load tests before using XSLT in production. If you need to use XSLT and if performance via JAXP is insufficient, you may consider using a vendor API directly and wrapping it in a utility component using the Façade pattern. You might also look into XSLTC, an XSLT compiler recently donated to the Apache Software Foundation by Sun

Microsystems. It enables you to compile XSLT stylesheets into Java classes called *translets*. More information on XSLTC is available at <http://xml.apache.org/xalan-j/xsltc/>.

2.2.2 JDOM

The first thing that stands out about this JAX family member is its lack of a JAX acronym. With JAXP now at your disposal, you can write parser-independent XML application code. However, there is another API that can simplify things even further. It is called the Java Document Object Model (JDOM), and has been recently accepted as a formal recommendation under the Java Community Process.

JDOM, created by Jason Hunter and Brett McLaughlin, provides a Java-centric API for working with XML data structures. It was designed specifically for Java and provides an easy-to-use object model already familiar to Java developers. For example, JDOM uses Java collection classes such as `java.util.List` to work with XML data like node-sets. Furthermore, JDOM classes are concrete implementations whereas the DOM classes are abstract. This makes them easy to use and removes your dependence on a specific vendor's DOM implementation, much like JAXP.

The most recent version of JDOM has been retrofitted to use the JAXP API. This means that your use of JDOM does not subvert the JAXP architecture, but builds upon it. When the JDOM builder classes create an XML object, they invoke the JAXP API if available. Otherwise, they rely on a default provider for parsing (Xerces) and a default XSLT processor (Xalan). The JDOM architecture is depicted in figure 2.9.

Table 2.7 lists the central JDOM classes. As you can see, they are named quite intuitively. JDOM documents can be created in memory or built from a stream, a file, or a URL.

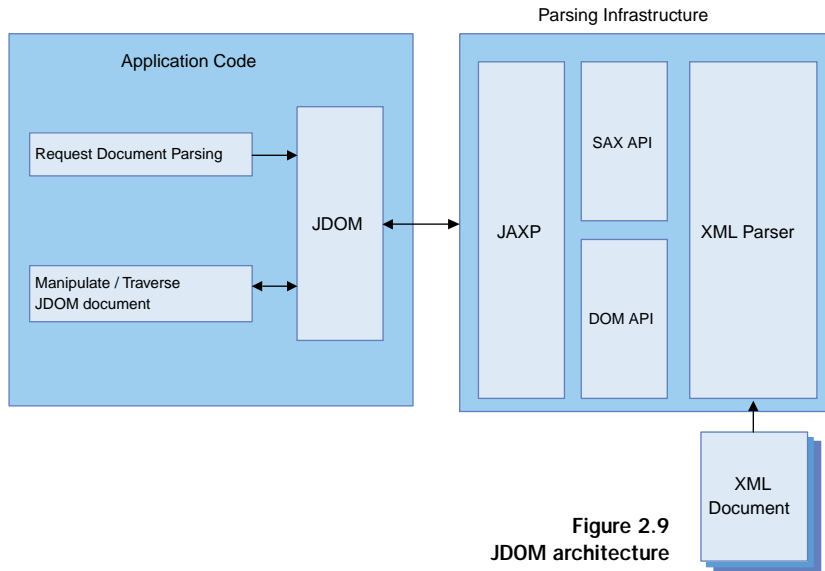


Figure 2.9 JDOM architecture

Table 2.7 Core JDOM classes

Class name	Description
org.jdom.Document	The primary interface to a JDOM document.
org.jdom.Element	An object representation of an XML node.
org.jdom.Attribute	An object representation of an XML node's attribute.
org.jdom.ProcessingInstruction	JDOM contains objects to represent special XML content, including application-specific processing instructions.
org.jdom.input.SAXBuilder	A JDOM builder that uses SAX.
org.jdom.input.DOMBuilder	A JDOM builder that uses DOM.
org.jdom.transform.Source	A JAXP XSLT Source for JDOM Documents. The JDOM is passed to the Transformer as a JAXP SAXSource.
org.jdom.transform.Result	A JAXP XSLT Result for JDOM Documents. Builds a JDOM from a JAXP SAXResult.

To quickly demonstrate how easy JDOM is to use, let us build our product catalog document from scratch, in memory, and then write it to a file. To do so, we simply build a tree of JDOM `Element`s and create a JDOM `Document` from it. The code to make this happen is shown in listing 2.10. When you compile and run this code, you should find a well-formatted version of the XML document shown in listing 2.1 in your current directory.

Listing 2.10 Building a document with JDOM

```
import org.jdom.*;
import org.jdom.output.XMLOutputter;
import java.io.FileOutputStream;

public class JDOMCatalogBuilder {

    public static void main(String[] args) {

        // construct the JDOM elements

        Element rootElement = new Element("product-catalog");
        Element productElement = new Element("product");

        productElement.addAttribute("sku", "123456");
        productElement.addAttribute("name", "The Product");

        Element en_US_descr = new Element("description");
        en_US_descr.addAttribute("locale", "en_US");
        en_US_descr.addContent("An excellent product.");

        Element es_MX_descr = new Element("description");
        es_MX_descr.addAttribute("locale", "es_MX");
        es_MX_descr.addContent("Un producto excelente.");

        Element en_US_price = new Element("price");
        en_US_price.addAttribute("locale", "en_US");
        en_US_price.addAttribute("unit", "USD");
        en_US_price.addContent("99.95");

        Element es_MX_price = new Element("price");
        es_MX_price.addAttribute("locale", "es_MX");
        es_MX_price.addAttribute("unit", "MXP");
        es_MX_price.addContent("9999.95");

        // arrange elements into a DOM tree

        productElement.addContent(en_US_descr);
        productElement.addContent(es_MX_descr);
        productElement.addContent(en_US_price);
        productElement.addContent(es_MX_price);

        rootElement.addContent(productElement);
        Document document = new Document(rootElement);

        // output the DOM to "product-catalog.xml" file
    }
}
```

Creates element attributes

Adds text to the element

Builds the DOM by adding one element as content to another

Wraps root element and processing instructions

```
XMLOutputter out = new XMLOutputter("  ", true); <← Indents element two
try {                                             spaces and uses newlines
    FileOutputStream fos = new FileOutputStream("product-catalog.xml");
    out.output(document, fos); <← Writes the JDOM representation to a file
} catch (Exception e) {
    System.out.println("Exception while outputting JDOM:");
    e.printStackTrace();
}
}
```

Due to its intuitive interface and support for JAXP, you will see JDOM used extensively in remaining chapters. You can find detailed information about JDOM and download the latest version from <http://www.jdom.org>.

2.2.3 JAXB

The Java API for XML Binding (JAXB) is an effort to define a two-way mapping between Java data objects and XML structures. The goal is to make the persistence of Java objects as XML easy for Java developers. Without JAXB, the process of storing and retrieving (*serializing* and *deserializing*, respectively) Java objects with XML requires the creation and maintenance of cumbersome code to read, parse, and output XML documents. JAXB enables you to work with XML documents as if they were Java objects.

DEFINITION *Serialization* is the process of writing out the state of a running software object to an output stream. These streams typically represent files or TCP data sockets.

The JAXB development process requires the creation of a DTD and a *binding schema*—an XML document that defines the mapping between a Java object and its XML schema. You feed the DTD and binding schema into a *schema compiler* to generate Java source code. The resulting classes, once compiled, handle the details of the XML-Java conversion process. This means that you do not need to explicitly perform SAX or DOM parsing in your application code. Figure 2.10 depicts the JAXB process flow.

Early releases of JAXB show improved performance over SAX and DOM parsers because its classes are lightweight and precompiled. This is a positive sign for the future of JAXB, given the common concerns about performance when using XML.

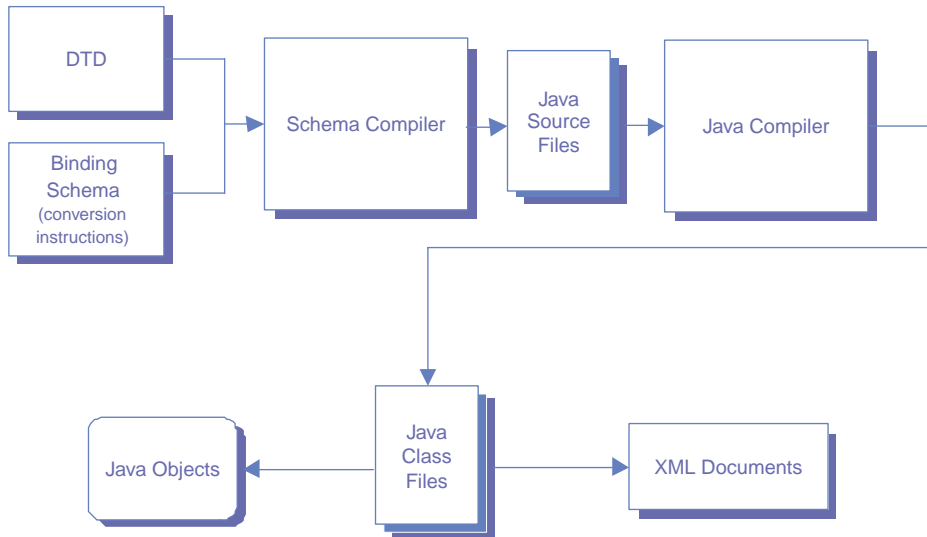


Figure 2.10 JAXB architecture

One tradeoff to consider before using JAXB is a loss of system flexibility, since any change in your XML or object structures requires recompilation of the JAXB classes. This can be inconvenient or impractical for rapidly evolving systems that use JAXB extensively. Each change to the JAXB infrastructure requires regenerating the JAXB bindings and retesting the affected portions of the system.

JAXB manifests other issues in its current implementation that you should explore before using it in your applications. For example, the process by which XML data structures are created from relational tables is overly simplistic and resource intensive. Issues such as these are expected to subside as the specification matures over time. We provide an example of using JAXB in the remainder of this section. More information about the capabilities and limitations of this API are available at <http://java.sun.com/xml/jaxb/>.

Binding Java objects to XML

To see JAXB in action, we turn once again to our product catalog example from listing 2.1. We previously developed the DTD corresponding to this document, which is shown in listing 2.2. Creating the binding schema is a bit more complicated. We start by creating a new binding schema file called `product-catalog.xjs`. Binding schemas in the early access version of JAXB always have the following root element:

```
<xml-java-binding-schema version="1.0-ea">
```

This element identifies the document as a binding schema. We now define our basic, innermost elements in the product-catalog document:

```
<element name="description" type="class">
  <attribute name="locale"/>
  <content property="description"/>
</element>
```

and

```
<element name="price" type="class">
  <attribute name="locale"/>
  <attribute name="unit"/>
  <content property="price"/>
</element>
```

The `type` attribute of the `element` node denotes that the elements of type `description` and `price` in the `product-catalog` document are to be treated as individual Java objects. This is necessary because both `description` and `price` have their own attributes as well as content.

The `content` element in each of the above definitions tells the JAXB compiler to create a property for the enclosing class with the specified `name`. The content of the generated `Description` class will be accessed via the `getDescription` and `setDescription` methods. Likewise, the `Price` class content will be accessed via methods called `getPrice` and `setPrice`.

Having described these basic elements, we can now refer to them in the definition of the `product` element.

```
<element name="product" type="class">
  <content>
    <element-ref name="description"/>
    <element-ref name="price"/>
  </content>
</element>
```

The `product` element maps to a Java class named `Product` and will contain two `Lists` as instance variables. One of these will be a `List` of `Description` instances. The other will be a `List` of `Price` instances. Notice the use of `element-ref` instead of `element` in the definition of the `description` and `price` nodes. This construct can be used to create complex object structures and to avoid duplication of information in the binding document.

The final element to bind is the root element, `product-catalog`. Its binding is defined as follows:

```
<element name="product-catalog" type="class" root="true">
  <content>
    <element-ref name="product"/>
  </content>
</element>
```

Notice the `root=true` attribute in this binding definition. This attribute identifies `product-catalog` as the root XML element. From this definition, the JAXB compiler will generate a class called `ProductCatalog`, containing a `List` of `Product` instances. The complete JAXB binding schema for our example is shown in listing 2.11.

Listing 2.11 Complete JAXB binding schema example

```
<xml-java-binding-schema version="1.0-ea">
  <element name="description" type="class">
    <attribute name="locale"/>
    <content property="description"/>
  </element>
  <element name="price" type="class">
    <attribute name="locale"/>
    <attribute name="unit"/>
    <content property="price"/>
  </element>
  <element name="product" type="class">
    <content>
      <element-ref name="description"/>
      <element-ref name="price"/>
    </content>
  </element>
  <element name="product-catalog" type="class" root="true">
    <content>
      <element-ref name="product"/>
    </content>
  </element>
</xml-java-binding-schema>
```

Now that we have a DTD and a binding schema, we are ready to generate our JAXB source code. Make sure you have the JAXB jar files in your classpath and execute the following command:

```
# java com.sun.tools.xjc.Main product-catalog.dtd product-catalog.xjs
```

If all goes well, you will see the following files created in your current directory:

```
Description.java  
Price.java  
Product.java  
ProductCatalog.java
```

You can now compile these classes and begin to use them in your application code.

Using JAXB objects

Using your compiled `JAXB` classes within your application is easy. To read in objects from XML files, you simply point your JAXB objects at the appropriate file and read them in. If you are familiar with the use of `java.io.ObjectInputStream`, the concept is quite similar. Here is some code you can use to read in the product catalog document via JAXB:

```
ProductCatalog catalog = null;  
File productCatalogFile = new File("product-catalog.xml");  
try {  
    FileInputStream fis  
        = new FileInputStream(productCatalogFile);  
    catalog = ProductCatalog.unmarshal(fis);  
} catch (Exception e) {  
    // handle  
}  
finally {  
    fis.close();  
}
```

To reverse the process and save the `ProductCatalog` instance as XML, you could do the following:

```
try {  
    FileOutputStream fos  
        = new FileOutputStream(productCatalogFile);  
    catalog.marshal(fos);  
} catch (Exception e2) {  
    // handle  
}  
finally {  
    fos.close();  
}
```

In the course of application processing, use your JAXB objects just as you would any other object containing instance variables. In many cases, you will need to iterate through the children of a given element instance to find the data you need. For example, to get the U.S. English description for a given `Product` instance `product`, you would need to do the following:

```
String description = null;  
List descriptions = product.getDescription();  
ListIterator it = descriptions.listIterator();
```



```

while (it.hasNext()) {
    Description d = (Description) it.next();
    if (d.getLocale().equals(en_US)) {
        description = d.getDescription();
        break;
    }
}

```

This type of iteration is necessary when processing XML data through all APIs, and is not specific to JAXB. It is a necessary part of traversing tree data structures like XML.

We invite you to explore the full capabilities of JAXB at the URL given near the beginning of this section. This can be a very useful API in certain applications, especially those with serious performance demands.

2.2.4 *Long Term JavaBeans Persistence*

Easily the most poorly named Java XML API, *Long Term JavaBeans Persistence* defines an XML mapping API for JavaBeans components. It is similar in function to JAXB, but leverages the JavaBeans component contract instead of a binding schema to define the mapping from Java to XML. Since JavaBeans must define *get* and *set methods* for each of their publicly accessible properties, it was possible to develop XML-aware components that can serialize JavaBeans to XML without a binding schema. These components use the Java reflection API to inspect a given bean and serialize it to XML in a standard format.

This API has become a part of the Java 2 Standard Edition as of version 1.4. There is no need to download any extra classes and add them to your classpath. The primary interfaces to this API are summarized in table 2.8. These classes behave in a similar fashion to `java.io.ObjectInputStream` and `java.io.ObjectOutputStream`, but use XML instead of a binary format.

Table 2.8 Core Long Term JavaBeans Persistence classes

Class name	Description
<code>java.beans.XMLEncoder</code>	Serializes a JavaBean as XML to an output stream.
<code>java.beans.XMLDecoder</code>	Reads in a JavaBean as XML from an input stream.

Writing a JavaBean to XML

As an example, let us define a simple JavaBean with one property, as follows:

```

public class SimpleJavaBean {
    private String name;

```

```
public SimpleJavaBean(String name) {
    setName(name);
}

// accessor
public String getName() { return name; }

// modifier
public void setName(String name) { this.name = name; }
}
```

As you can see, this bean implements the JavaBeans contract of providing an accessor and modifier for its single property. We can save this bean to an XML file named `simple.xml` using the following code snippet:

```
import java.beans.XMLEncoder;
import java.io.*;

...

XMLEncoder e
    = new XMLEncoder(new BufferedOutputStream(
        new FileOutputStream("simple.xml")));
e.writeObject(new SimpleJavaBean("Simpleton"));
e.close();
```

The code above creates an `XMLEncoder` on top of a `java.io.BufferedOutputStream` representing the file `simple.xml`. We then pass the `SimpleJavaBean` instance reference to the encoder's `writeObject` method and close the stream. The resulting file contents are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.0" class="java.beans.XMLDecoder">
  <object class="SimpleJavaBean">
    <void property="name">
      <string>Simpleton</string>
    </void>
  </object>
</java>
```

We will not cover the XML syntax in detail, since you do not need to understand it to use this API. Detailed information about this syntax is available in the specification, should you need it.

Restoring a JavaBean from XML

Reading a previously saved JavaBean back into memory is equally simple. Using our `SimpleJavaBean` example, the bean can be reinstated using the following code:

```
XMLDecoder d
    = new XMLDecoder( new BufferedInputStream(
        new FileInputStream("simple.xml")));
SimpleJavaBean result = (SimpleJavaBean) d.readObject();
d.close();
```

The `XMLDecoder` knows how to reconstitute any bean saved using the `XMLEncoder` component. This API can be a quick and painless way to export your beans to XML for use by other tools and applications. And remember, you can always transform the bean's XML to another format via XSLT to make it more suitable for import into another environment.

2.2.5 JAXM

The Java API for XML Messaging (JAXM) is an enterprise Java API providing a standard access method and transport mechanism for SOAP messaging in Java. It currently includes support for the SOAP 1.1 and SOAP with Attachments specifications. JAXM supports both synchronous and asynchronous messaging.

The JAXM specification defines the various services that must be provided by a JAXM implementation provider. Using any compliant implementation, the developer is shielded from much of the complexity of the messaging system, but has full access to the services it provides. Figure 2.11 depicts the JAXM architecture.

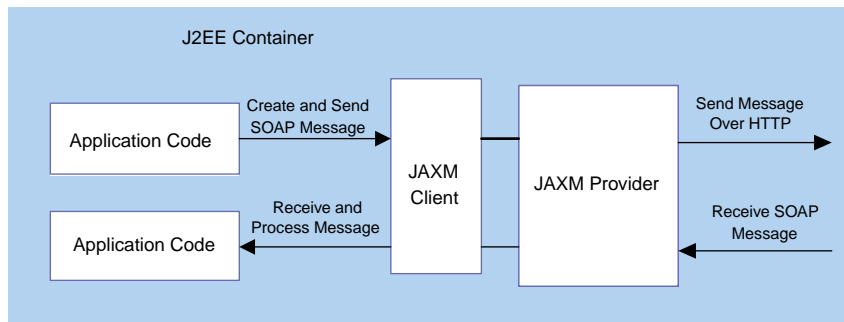


Figure 2.11 JAXM architecture

The two main components of the JAXM architecture are the JAXM Client and Provider. The Client is part of the J2EE Web or EJB container that provides access to JAXM services from within your application. The Provider may be implemented in any number of ways and is responsible for sending and receiving SOAP messages. With the infrastructure in place, sending and receiving SOAP messages can be done exclusively through the JAXM API.

The JAXM API consists of two packages, as summarized in table 2.9. Your components access JAXM services via a `ConnectionFactory` and `Connection` interface, in the same way you would obtain a handle to a message queue in the Java Messaging Service (JMS) architecture. After obtaining a `Connection`, you can use it to create a structured SOAP message and send it to a remote host via HTTP(S). JAXM also provides a base Java servlet for you to extend when you need to handle inbound SOAP messages.

Table 2.9 The JAXM API packages

Package name	Description
<code>javax.xml.messaging</code>	Contains the <code>ConnectionFactory</code> and <code>Connection</code> interfaces and supporting objects.
<code>javax.xml.soap</code>	Contains the interface to the SOAP protocol objects, including <code>SOAPEnvelope</code> , <code>SOAPHeader</code> , and <code>SOAPBody</code>

At the time of this writing, JAXM 1.0.1 is available as part of the Java XML Pack and is clearly in the lead of all APIs under development in terms of standardizing the transmission of SOAP messages in Java. Since the creation and consumption of SOAP messages is a complex topic, we defer an example of using JAXM to chapter 4. There we use JAXM to create and access web services in J2EE.

More information about JAXM can be found at <http://java.sun.com/xml/jaxm/>. Details about the Java XML Pack can be found at <http://java.sun.com/xml/javaxmlpack.html>.

2.2.6 JAX-RPC

JAX-RPC is a Java-specific means of performing remote procedure calls using XML. JAX-RPC implements the more general XML-RPC mechanism that is the basis of SOAP. Using JAX-RPC, you can expose methods of the beans running in your EJB container to remote Java and non-Java clients. An early access release of the JAX-RPC is now available as part of the Java XML Pack. Up-to-date details about JAX-RPC are at <http://java.sun.com/xml/jaxrpc/>.

It should be noted that SOAP is fast becoming the preferred method of implementing XML-RPC for web services. Since JAXM already implements the SOAP protocol and has a more mature reference implementation available, the future of the JAX-RPC API remains somewhat uncertain.

2.2.7 JAXR

A critical component to the success of web services is the ability to publish and access information about available services in publicly available registries. Currently, there are several competing standards in the area of web services registries. UDDI and ebXML Registry are currently the two most popular of these standards.

To abstract the differences between registries of different types, an effort is underway to define a single Java API for accessing any type of registry. The planned result is an API called the Java API for XML Registries (JAXR). JAXR will provide a layer of abstraction from the specifics of each registry system, enabling standardized access to web services information from Java.

JAXR is expected to handle everything from executing complex registry queries to submitting and updating your own data to a particular registry system. The primary benefit is that you will have access to heterogeneous registry content without having to code your components to any specific format. Just as JNDI enables dynamic discovery of resources, JAXR will enable dynamic discovery of XML-based registry information. More information on JAXR is available at <http://java.sun.com/xml/jaxr/>.

The JAXR specification is currently in public review draft, and an early access reference implementation is part of the Java XML Pack. Because of its perceived future importance with regard to web services and the number of parties interested in ensuring its interface is rock solid, this specification is likely to change dramatically before its first official release. We encourage you to stay on top of developments in this API, especially if you plan to produce or consume web services in J2EE.

2.3 *Summary*

The chapter has been a whirlwind tour of current XML tools and technologies, along with their related Java APIs. Now that you are familiar with the state and direction of both XML and J2EE, we can begin to use them together to enhance your application architecture.

By now, you should be comfortable with viewing XML as a generic metalanguage and understand the relationships between XML, XML parsers, XSLT processors, and XML-based technologies. You should also understand how XML is validated and constrained at high level. Perhaps most importantly, you should see how the various pieces of XML technology fit together to enable a wide

variety of functionality. You will see many of the technologies and APIs discussed in this chapter implemented by the examples in the remaining chapters.

Of all the topics covered in this chapter, web services is by far the hottest topic in business application development today. Chapter 4 contains the details you need to implement and consume web services in J2EE. Chapter 6 provides an end-to-end example of using web services via a case study.



Application development

This chapter

- Demonstrates the use of XML-based component interfaces
- Discusses XML data persistence options
- Identifies important XML technologies at the application logic layer

This chapter is about enhancing the internal structure of your J2EE applications with select XML technologies. We demonstrate the use of XML interfaces between components and discuss the potential advantages and drawbacks of taking this approach. We use the Value Object design pattern and provide a detailed example to illustrate the implementation of this XML interface technique.

In the second part of the chapter, we examine the use of XML as a persistent representation of your application data. We take an in-depth look at emerging XML data storage and retrieval technologies, including XQuery, PDOM, and XQL. We highlight the potential advantages and disadvantages of using XML technology for data persistence and examine the maturity level of current implementations of each technology.

Finally, we examine some options for translating between relational and XML data representations using the Data Access Object pattern. The examples demonstrate both an application-specific and a generic approach to bridging the gap between relational JDBC data sources and XML data.

3.1 *XML component interfaces*

A *component interface* refers to the representation of data within your application. For example, what does your customer component look like? How can its data be accessed and manipulated? An XML component interface uses XML to represent this information. Throughout this section, we will examine the advantages and disadvantages of using XML within your application components.

XML receives most of its attention for its potential to integrate applications, enterprises, and industries via self-describing data. These data can be validated and manipulated in generic ways using generic tools, and detailed grammars can be created to standardize and enforce the XML dialects spoken between systems.

However, the benefits of XML technology reach far beyond systems integration. In chapter 5, you will see that XML tools can be used to serve customized user views of application data through technologies like XSLT and XSP. In this chapter, we expand our view of XML as an application development tool to include internal application structure and data representation. In many instances, XML can be used as the native data format for your entire application. For example, using XML to represent customer, order, and product data allows you to create a standard format that can be reused across applications. Your customer relationship management system can then use the same XML components as your e-commerce application. Additionally, these data can be

persisted in their native XML format or converted to a relational format for storage in an RDBMS.

To understand how XML can be used as an internal data format, we must distinguish between the XML data structures in your application's memory space and the concept of an XML document. The term document conjures images of a static file located on a file system. In fact, your application has little interest in such documents. Your application holds its data resident in memory, passing it from one component to the next and operating on it. At some point, this data may or may not be persisted to a storage medium, which could be a file (document) or a database. See figure 3.1.

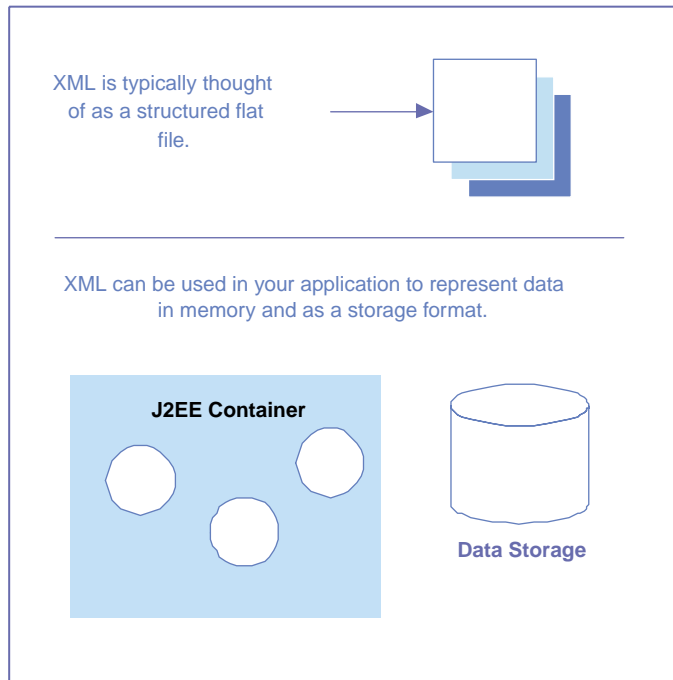


Figure 3.1 Viewing XML as more than a flat file

Proprietary formats vs. XML value objects

A *value object* is an in-memory representation of data that is suitable for passing between tiers of your application. These objects are often implemented as proprietary software components. For example, your application might employ a value object called `CustomerData` to represent customer information.

It is just as easy, and in many cases more convenient, to use an XML DOM value object to hold that customer information.

Using an XML DOM object instead of a proprietary object has several advantages.

- You can access and manipulate a DOM using standard XML tools and APIs.
- Your application data is ready to be transformed into virtually any output format via XSLT.
- You can expose your component interfaces to external applications that have no knowledge of your proprietary data objects.
- Using XML at this level provides a great deal of flexibility and ensures loose coupling between your components and the clients that invoke them.

3.1.1 *Using value objects*

The use of value objects is described generically in the Value Object design pattern in appendix A. In this pattern, a serializable utility object is used to pass data by value between remote components using RMI. In this section, we will compare a simple implementation of that pattern using a proprietary value object with an implementation using an XML value object.

An example scenario

To analyze the concepts covered in this chapter, we provide an example. The application we use is an ordering system. It contains customer information such as address and phone number, order information, and product data. The value objects that represent these data are straightforward components that can demonstrate the use of XML in the application logic layer.

A proprietary value object implementation

The first component that we create is the class to represent a customer in our application. Using the traditional J2EE approach, you might construct a `CustomerValue` object as shown in listing 3.1.

Listing 3.1 A value object for customer data

```
import java.io.Serializable;

/**
 * Value object for passing
 * customer data between remote
 * components.
 */
public class CustomerValue
```

```
implements Serializable {  
    /** Customer ID can't be changed */  
    private long customerId;  
  
    public String firstName;  
    public String lastName;  
    public String streetAddress;  
    public String city;  
    public String state;  
    public String zipCode;  
    public String phoneNumber;  
    public String emailAddress;  
  
    public CustomerValue(long id) {  
        customerId = id;  
    }  
  
    public long getCustomerId() {  
        return customerId;  
    }  
}
```

Customer
data

This is a simple object that encapsulates our customer data. One benefit of using a proprietary object to represent a customer is that you can implement validation logic specific to the customer data if necessary. However, using this custom object also has two major drawbacks. First, this object cannot be reused to represent any other type of data in your application (e.g., an order). Thus, you will have to create and maintain many types of value objects and have many specialized interfaces between your components.

The second drawback of using this proprietary object is that any client receiving a `CustomerValue` object must know what it is and how to access its data specifically. The client must know the difference between a `CustomerValue` and an `OrderValue` at the interface level and treat them differently. This tightly couples the client and server components, creates code bloat on both sides, and severely hampers the flexibility of the system.

Overcoming limitations with XML value objects

XML data structures can overcome the limitations of proprietary object implementations because they present a generic interface to the data they encapsulate. A DOM object is used in exactly the same manner regardless of its contents. Client components need not worry about reflecting or casting of objects to specific types, and need access to only the XML API classes to handle any type of data. Additionally, most of the validation logic for a certain

type of data can be enforced generically using validating parsers and a DTD or XML Schema.

Figure 3.2 shows the `CustomerValue` object represented as a DOM tree instead of a proprietary object. Note that the DOM makes it easy to add more structure to the customer data, encapsulating the address fields within a node. To accomplish this with proprietary objects, we would need to write an `AddressValue` object and add its reference to the `CustomerValue` object in listing 3.2. The more structure we add, the more code is required in the proprietary approach. This is not true with XML.

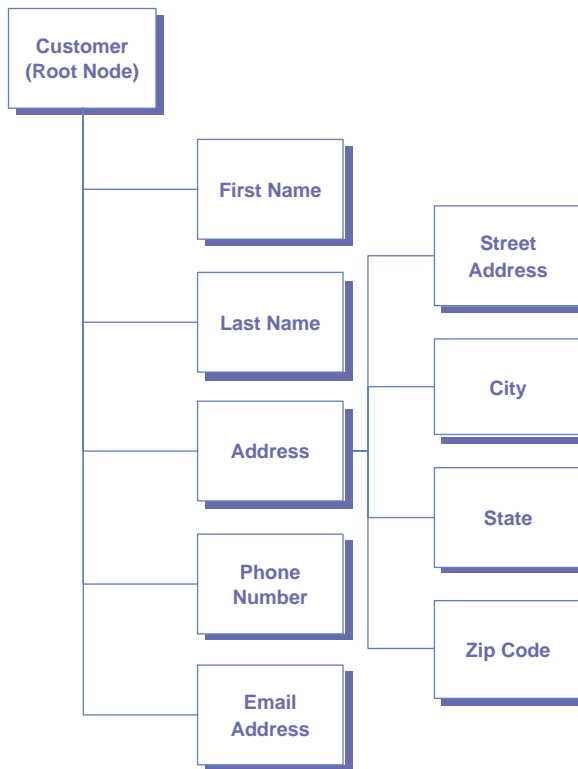


Figure 3.2
Customer data represented
using a DOM tree

Listing 3.2 shows what the customer DOM might look like if it were serialized out to a file. However, based on the requirements of your application, it is possible that this data could be transient and never be stored in a file.

Remember to keep the concepts of an XML data structure and an XML file separate in your mind.

Listing 3.2 Customer XML data serialized to a file

```
<?xml version="1.0"?>
<!-- Reference to a DTD to validate our customer data -->
<!DOCTYPE customer SYSTEM "http://www.example.com/customer.dtd">
<customer id="123456">
  <first-name>John</first-name>
  <last-name>Doe</last-name>
  <address>
    <street>123 Main</street>
    <city>Anytown</city>
    <state>CA</state>
    <zip>99999</zip>
  </address>
  <phone>800-555-9999</phone>
  <email-address>john@doe.com</email-address>
</customer>
```

It is clearly beneficial to use XML in our component interfaces from the standpoints of flexibility and reusability. Though our discussion used a simple example, the concepts can be applied to larger systems where the advantages of an XML approach become even more evident.

3.1.2 Implementing XML value objects

Now that we have chosen to use XML for our internal data representation, let's walk through a more robust example using the value object approach. For purposes of this implementation, we'll use the JDOM API. Later in this section, we will discuss the use of JDOM over DOM in this setting. As we discussed in chapter 2, JDOM is layered on top of the DOM and SAX APIs, as well as specific parser and XSLT engines, to provide a Java-friendly way to use XML structures. Here we use JDOM to create new XML data structures, manipulate them, and share them with clients.

The requirements for this example are simple but sufficient for our purposes. We are required to retrieve detailed customer information from an enterprise data source based on the customer's unique identifier. To accomplish this, we use the Data Access Object design pattern. In this pattern, the data access object (DAO) hides the complexity of interacting with a persistent

data source and provides a simple interface for other components to use. The Data Access Object pattern is discussed in detail in appendix A.

To implement this pattern, we use an EJB session bean called the `CustomerDataBean`. This bean will obtain the customer data using a `CustomerDAO` (data access object), which obtains customer data in their raw format from a JDBC data source, converts them to XML using JDOM, and returns them to the `CustomerDataBean`. The `CustomerDataBean` then returns the JDOM Document to the remote caller. This scenario is depicted in figure 3.3.

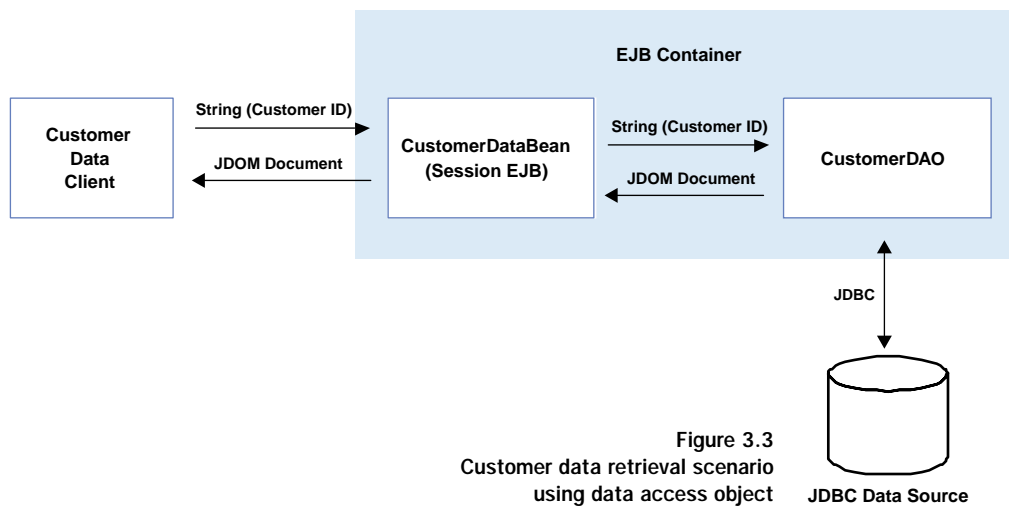


Figure 3.3
Customer data retrieval scenario
using data access object

JDBC Data Source

The CustomerDataBean

First, we implement the `CustomerDataBean` session EJB. This bean declares an instance variable to hold a reference to its `CustomerDAO` helper object.

```
public transient CustomerDAO cDAO;
```

At creation time, the bean obtains a reference to the JDBC data source using JNDI and instantiates its `CustomerDAO` object.

```
protected void buildDAO() throws EJBException {
    try{
        javax.naming.Context jndiCtx
            = new javax.naming.InitialContext();
        javax.sql.DataSource ds = (javax.sql.DataSource)
            jndiCtx.lookup("java:comp/env/jdbc/CustomerDB");
        cDAO = new CustomerDAO(ds);
    } catch (Exception e) {
```

```

        throw new EJBException(e);
    }
}

```

Then, when invoked by a remote client, the bean obtains the requested customer data in XML format from the `CustomerDAO` and returns it to the caller.

```

public org.jdom.Document getCustomerInfo(String customerId)
    throws CustomerNotFoundException {
    Document custData = cDAO.getCustomerInfo(customerId);
    return custData;
}

```

The complete code for the `CustomerDataBean` is shown in Listing 3.3.

Listing 3.3 Implementation of the `CustomerDataBean`

```

import javax.sql.DataSource;
import javax.ejb.EJBException;

import org.jdom.Document;

/**
 * A session bean that retrieves customer
 * data as a JDOM Document
 */
public class CustomerDataBean implements javax.ejb.SessionBean {

    public javax.ejb.SessionContext ctx;

    // transient so it won't be
    // serialized on passivation
    public transient CustomerDAO cDAO;

    public void ejbCreate() {
        buildDAO(); ← Retrieves data
                    | from DAO
    }

    /**
     * Get a JDOM Document containing the specified
     * customer's information.
     * @param customerId Unique ID of the customer
     * @return JDOM containing the customer information
     * @throws CustomerNotFoundException
     */
    public org.jdom.Document getCustomerInfo(String customerId)
        throws CustomerNotFoundException {
        Document custData = cDAO.getCustomerInfo(customerId);
        return custData;
    }

    protected void buildDAO() throws EJBException {
        // look up data source in environment
        // and pass to the data access object's
    }
}

```

Gets JDOM Document from DAO


```

// constructor
try{
    javax.naming.Context jndiCtx = new javax.naming.InitialContext();
    javax.sql.DataSource ds = (javax.sql.DataSource)
    jndiCtx.lookup("java:comp/env/jdbc/CustomerDB");
    cDAO = new CustomerDAO(ds); ← Passes data source
    } catch (Exception e) {           to DAO constructor
        throw new EJBException(e);
    }
}

public void ejbRemove() { }

// restore Data Access Object when activated
public void ejbActivate() { buildDAO(); }

public void ejbPassivate() { }

public void setSessionContext(javax.ejb.SessionContext ctx) {
    this.ctx = ctx;
}
}

```

As you can see, the `CustomerDataBean` is acting as a proxy between the data access object and remote clients in this example. In practice, the `CustomerDataBean` would probably cache the `Document` retrieved from the `CustomerDAO` object for use in subsequent requests.

The Customer data access object

The interesting code in the `CustomerDAO` class is the `getCustomerInfo` method, which performs all the relational-to-XML data translation. After executing a prepared statement, this method creates a new JDOM `Document` to hold the results.

```

Element root = new Element("customer");
    doc = new Document(root);

```

Various customer data fields are then added to the document. For simplicity, we use elements for each field. XML attributes could be used to hold nonforeign key values just as easily.

```

// first name
Element fnElement =
    root.addContent(new Element("first-name"));
fnElement.addContent(rs.getString("FIRST_NAME"));
...

```

After all the fields have been created and populated, the complete JDOM document is returned to the caller, our session bean in this case. Listing 3.4 contains the implementation code for this data access object.

Listing 3.4 Implementation of the CustomerDAO class

```
import org.jdom.Document;
import org.jdom.Element;

import javax.sql.DataSource;
import java.sql.*;

/**
 * A Data Access Object
 * for customer data
 */
public class CustomerDAO {

    protected DataSource ds = null;

    private final static String GET_CUST_SQL =
        "select * from customers where custId=?";

    public CustomerDAO(DataSource ds) {
        this.ds = ds;
    }

    /** Return customer data as a JDOM Document */
    public Document getCustomerInfo(String customerId)
        throws CustomerNotFoundException {
        Document doc = null;
        Connection con = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            con = ds.getConnection();
            ps = con.prepareStatement(GET_CUST_SQL);
            ps.setString(1, customerId);
            rs = ps.executeQuery();
            // only one row
            rs.next();

            // build a JDOM Document from the ResultSet
            // -----
            Element root = new Element("customer");
            doc = new Document(root);
            // first name
            Element fnElement =
                root.addContent(new Element("first-name"));
            fnElement.addContent(rs.getString("FIRST_NAME"));

            // last name
```

SQL to retrieve customer info from database

Connects to data source and retrieves the data

Builds customer JDOM Document

```

Element lnElement =
    root.addContent(new Element("last-name"));
lnElement.addContent(rs.getString("LAST_NAME"));

// address info
Element address
    = root.addContent(new Element("address"));
Element streetElement =
    address.addContent(new Element("street"));
streetElement.addContent(rs.getString("STREET"));
Element cityElement =
    address.addContent(new Element("city"));
cityElement.addContent(rs.getString("CITY"));
Element stateElement =
    address.addContent(new Element("state"));
stateElement.addContent(rs.getString("STATE"));
Element zipElement =
    address.addContent(new Element("zip"));
stateElement.addContent(rs.getString("ZIP"));

// phone number
Element phElement =
    root.addContent(new Element("phone"));
phElement.addContent(rs.getString("PHONE"));

// email address
Element emElement =
    root.addContent(new Element("email-address"));
emElement.addContent(rs.getString("EMAIL"));

} catch (Exception e) {
    throw new CustomerNotFoundException(customerId, e);
} finally {
    if (rs != null)
        try { rs.close(); } catch (SQLException sqle1) {}
    if (ps != null)
        try { ps.close(); } catch (SQLException sqle2) {}
    if (con != null)
        try { con.close(); } catch (SQLException sqle3) {}
}

// return a JDOM Document
return doc;
}

// other methods here to create and update customers
}

```

The `CustomerDAO` implementation shows just how simple it can be to create and use XML data structures in your application instead of proprietary objects.

The document returned by the `CustomerDAO` can now be easily transformed and used by remote clients generically via XML APIs and tools.

This example combines the Value Object pattern with the Data Access Object pattern to encapsulate the translation work between XML and non-XML data representations. One problem remains with the `CustomerDAO`, however. It is specific to translating customer data. A separate object would be required to translate other types of information, such as orders and invoices. Later in this chapter, we develop a more generic data access object that can translate between XML and relational data formats in a more general manner.

Using JDOM vs. DOM document interfaces

At the time of this writing, JDOM is not yet a standard Java or J2EE API. Although it will likely be added to the standard APIs in some form in the future, you may be hesitant to expose JDOM-based APIs to your application clients for now. Not to worry, JDOM also provides an easy way to output a more general DOM structure from a JDOM document. If we prefer to provide an `org.w3c.Document` interface to remote clients in our example, we simply add a few lines to the `CustomerDataBean` and change the return value for the `getCustomerInfo` business method. This means importing three more classes and altering the `getCustomerInfo` method slightly.

```
org.jdom.Document custData
    = cDAO.getCustomerInfo(customerId);
DOMOutputter outputter = new DOMOutputter();
try {
    return outputter.output(custData);
} catch (JDOMException je) {
    // handle conversion error
}
```

The “pure DOM” interface approach is shown in Listing 3.5.

Listing 3.5 Exposing the `org.w3c.Document` interface

```
import javax.sql.DataSource;
import javax.ejb.EJBException;

import org.jdom.JDOMException;
import org.jdom.output.DOMOutputter;

/**
 * A session bean that retrieves customer
 * data as a W3C DOM Document
 */
public class CustomerPureDOMDataBean implements javax.ejb.SessionBean {
```

```

public javax.ejb.SessionContext ctx;

// transient so it won't be
// serialized on passivation
public transient CustomerDAO cDAO;

public void ejbCreate() {
    buildDAO();
}

/**
 * Get a JDOM Document containing the specified
 * customer's information.
 * @param customerId Unique ID of the customer
 * @return JDOM containing the customer information
 * @throws CustomerNotFoundException
 */
public org.w3c.dom.Document
getCustomerInfo(String customerId) ← Uses DOMOutputter
    throws CustomerNotFoundException {
    org.jdom.Document custData = cDAO.getCustomerInfo(customerId);
    DOMOutputter outputter = new DOMOutputter(); ← Converts JDOM
    try {
        return outputter.output(custData);
    } catch (JDOMException je) {
        // handle conversion error
    }
    return null;
}

protected void buildDAO() throws EJBException {
    // look up data source in environment
    // and pass to the data access object's
    // constructor
    try{
        javax.naming.Context jndiCtx
            = new javax.naming.InitialContext();
        javax.sql.DataSource ds =
            (javax.sql.DataSource)
                jndiCtx.lookup("java:comp/env/jdbc/CustomerDB");
        cDAO = new CustomerDAO(ds);
    } catch (Exception e) {
        throw new EJBException(e);
    }
}

public void ejbRemove() { }

// restore Data Access Object when activated
public void ejbActivate() { buildDAO(); }

public void ejbPassivate() { }

```

```
public void setSessionContext(javax.ejb.SessionContext ctx) {  
    this.ctx = ctx;  
}  
}
```

Using JDOM vs. JAXB

You might be wondering why we chose JDOM over JAXB for our value object example. JAXB is, after all, a member of the Java XML extension APIs. JDOM is only a JCR at this point. The reason is one of flexibility.

Using a JDOM approach, very few of the objects in our system are tied to the internal structure of the value objects. When we alter the XML data structure, only those components that operate on the structures need to be changed, and then only if those objects are working with the XML data at the lowest level (e.g., traversing and populating nodes in code). None of the interfaces in the system need to change as a result of XML structure changes, reducing the amount of running code in the system that needs to be retested.

Using JAXB requires a tight coupling between the value objects and the data structure. XML structural changes require rebinding of the JAXB classes and retesting all of the components that interact with the JAXB objects.

3.1.3 When not to use XML interfaces

This book is about using XML in your J2EE applications with discretion. This entire section discusses the merits of using XML throughout your application as an internal data format. We would be remiss not to emphasize the fact that the above approach is not appropriate in certain circumstances. You need to consider some specific aspects of your system carefully before jumping in to an all-XML system.

Two of the most important considerations for deciding if XML is right for your component pertain to resource usage and performance.

XML component interfaces and resource usage

One major drawback in using DOM-based XML APIs is that the entire XML structure is present in memory whenever a DOM exists. If you have very large XML data structures, numerous instances of data structures, or both, you should estimate the amount of memory that will be consumed by your application at various load levels. You may find that passing DOM trees around inside your application is not feasible given the amount of data you plan to be processing simultaneously.

XML component interfaces and performance

Using JDOM (as shown in section 3.1.2) could result in slower response time due to the processing required to translate between data formats. While this bit of extra processing may not be a concern, the number of steps required to service a single request should always be considered since it can have a significant impact when aggregated across many simultaneous requests. Performance concerns become much more significant if you need to parse files when building your XML data tree. JDOM does allow the use of SAX to speed the process, but parsing may still take more time than you can afford in some real-time user-driven applications.

The point of this section is not to scare you away from using XML in application internals but rather to make you aware of the risks involved in doing so. It is up to you as the system architect to determine the balance between the flexibility and generality of XML and the performance and resource utilization needs of your application.

3.2 *XML and persistent data*

Data storage undoubtedly conjures up images of your relational database or ERP system. In some cases, however, storing your data persistently in XML format can be advantageous. This may be true if your application is managing a large repository of data that is file-based. An example of this might be an application that manages data feeds from partners using RosettaNet PIPs. The data being operated on is document-based, and the format of those documents is XML. In such cases, it may not make sense to translate the XML data into a relational format and store them in a database, unless there are other requirements that dictate so. In the future, you may actually be using an XML database product instead of a relational one, making the translation issue irrelevant.

Configuration data is another situation in which storing data in XML format is appropriate. Many applications now use XML to persistently store configuration parameters. Some even implement business rules and data validation logic via XML constructs. Since these data are relatively static, there is no need to store them in a database.

Clearly there are some situations in which the persistent storage of XML is appropriate. This fact presents some interesting challenges. Specifically, as the amount of XML data grows, finding and retrieving each specific piece of data becomes more challenging. Integrating data from separate XML documents is even more challenging. Several related efforts are currently underway at the W3C and other organizations to define a standard mechanism for querying

XML data efficiently. The W3C is in the process of defining a standard called XQuery, which we examine in the next section.

Another issue with XML data storage is one of performance and resource utilization. XML is a necessarily verbose text format that arranges data in a tree. This means that XML files are usually much larger than the data they contain, and that they can be slow to search. Reading a large XML document into memory can be impossible at times, rendering the DOM approach useless for large XML repositories. Technologies are currently under development to address these nonfunctional XML data persistence requirements. Technologies such as the Persistent Document Object Model (PDOM) are being developed to optimize XML file storage and enable faster XML searching mechanisms. We look at PDOM in section 3.2.2.

3.2.1 Querying XML data

Having your data locked up in XML documents is relatively useless if you cannot effectively locate, combine, and derive from those data in meaningful ways. Many groups of developers recognized this problem early in the development of XML, and a number of query languages and technologies have been developed to solve it. In this section, we examine the W3C attempt to unify these query technologies into a single, standard mechanism called XQuery.

XQuery

XQuery is a set of standard specifications currently under development by the W3C for querying XML data structures. When fully specified, it is intended to be for XML what SQL and stored procedures are for relational databases. You will use XQuery in your applications to locate, group, and join data from one or more XML data sets. Additionally, you will be able to use XQuery to derive new data sets and data types from existing XML sources.

At the time of this writing, the XQuery 1.0 specification is in a draft status, and many of the details are likely to change. XQuery focuses exclusively on the manipulation of XML data sets, and does not address nonfunctional issues such as performance, file management, or resource utilization. Due to its youth and lack of finalization, there are currently no enterprise tools that implement XQuery 1.0. We provide an overview here of what is to come because it is likely to become an important part of XML technology. Your J2EE data-aware objects will likely use XQuery to operate on XML data in the future.

There are several distinct parts of the XQuery 1.0 specification. These are summarized in table 3.1. These related specifications are intended to provide a complete definition of the data model, semantics, and syntax of the XQuery

language. XQuery is closely related to XML Schema, using the same data definition mechanisms and built-in types. It is also very closely tied to the XPath standard, and has even caused the XPath specification itself to be enhanced.

Table 3.1 The XQuery 1.0 specification set

XQuery specification	Contents
XML Query Requirements	Describes the generalized requirements for XQuery technology.
XML Query Use Cases	Contains use cases for the XQuery requirements.
XQuery 1.0 and XPath 2.0 Data Model	Describes the hierarchical XML data model shared by XPath and XQuery.
XQuery 1.0 Formal Semantics	Provides a formal description of terminology and mechanisms employed by Xquery.
XQuery 1.0: An XML Query Language	Describes the human-readable, expression-based form of Xquery.
XML Syntax for XQuery 1.0 (XQueryX)	Describes an XML-based variant of the XQuery language.

XQuery 1.0 is a human-readable, expression-based language built on concepts borrowed from many other languages, including SQL, XQL, and Object Query Language (OQL). There is also an XML-based variant of XQuery under development called XQueryX. We focus on the human-readable XQuery in this section.

Table 3.2 Types of XQuery 1.0 expressions

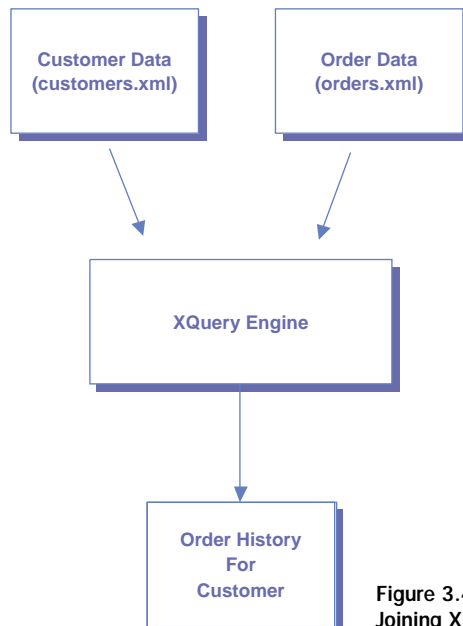
XQuery expression type	Description
Path expressions	An XPath string representing a specific node or set of nodes in an XML data tree. For example, //customers would return a set of all the customer nodes found in a document.
Element constructors	Templates for generating derived XML nodes by executing XQL statements. These are basically XML nodes with embedded XQL expressions that generate derived data when executed by an XQuery engine.
FLWR expressions	SQL-like structured statements containing some combination of FOR, LET, WHERE, and RETURN clauses. (Pronounced flower.)

(continued on next page)

Table 3.2 Types of XQuery 1.0 expressions (*continued*)

XQuery expression type	Description
Operators and functions	XQuery supports mathematical expressions, built-in functions such as text() and not(), as well as user-defined functions and function libraries.
Conditional expressions	XQuery supports an IF-THEN-ELSE construct for execution branching.
Quantified expressions	XQuery supports partial node set selection using the SOME keyword, and complete node set selection using the EVERY keyword.
Data type expressions	XQuery supports data type testing and modification expressions

One interesting feature of XQuery is that it contains several types of expressions that can be nested within one another in virtually any combination. The types of expressions available are summarized in table 3.2. The ability to nest these expressions within each other makes performing complex operations on XML data sets amazingly straightforward. For example, you might use XQuery to create new XML structures by joining existing XML documents. One such scenario is depicted in figure 3.4, which shows customer data and order data being joined to create an XML order history data set for a specific customer.

Figure 3.4
Joining XML data using XQuery

We create the `companies.xml` and `orders.xml` files in order to demonstrate an XQuery. The `customers.xml` file contains the following node:

```
<customer customer-id=123456>
  <customer-name>John Smith</customer-name>
</customer>
```

The `orders.xml` file contains the following node:

```
<order order-id=56789 customer-id=123456>
  <order-date>01-01-2001</order-date>
  <order-total>$59.00</order-total>
</order>
```

The query to accomplish the join might look something like this:

```
<order-history>
  {
    FOR
      $c in
        document(customers.xml)//customer[customer-id = 123456],
      $o in
        document(orders.xml)//order[customer-id = $c/customer-id]
    RETURN
      <customer-orders>
        {
          $c/customer-name,
          $o/order-date,
          $o/order-total
        }
      </customer-orders>
      SORT BY(order-date)
    }
  }
</order-history>
```

Builds query | ①

Retrieves results | ②

③ Outputs node-set

- ① First, this query looks for all customers in the `customers.xml` file with the attribute `customer-id` equal to 123456 and stores the result in the `$c` variable.
- ② Next, it retrieves all of the orders from the `order.xml` file with the `customer-id` attribute equal to 123456 and stores them in `$o`.
- ③ Finally, the resulting node contains the customer name, order date, and order total.

Given our sample data, the following XML node is the result of our query.

```
<customer-orders>
  John Smith, 01-01-2001, $59.00
</customer-orders>
```

You should be able to appreciate the potential power and usefulness of XQuery as a tool for searching existing data and deriving new data representations in XML. If you plan to use XML as a persistent storage mechanism, you can keep up to date on the latest XQuery developments at <http://www.w3c.org/XML/Query>.

To reiterate, XQuery is currently in draft status and no implementations are currently available. So what are your options for querying XML data today? Basically, your choices consist of using one of the query languages on which XQuery is based. These include Quilt, XML-QL, and XQL, among others. While none of these is nearly as sophisticated as XQuery intends to be, they can be sufficient for performing simple queries.

Querying XML using DAO and XQL

In this section, we develop an XML-aware data access object that uses XQL. This object provides the same functionality as the `CustomerDAO` from section 3.1, but obtains its data from an XML document instead of a relational database. Figure 3.5 depicts the result of our example. Following the figure, we walk you through the creation of the code for the data access object.

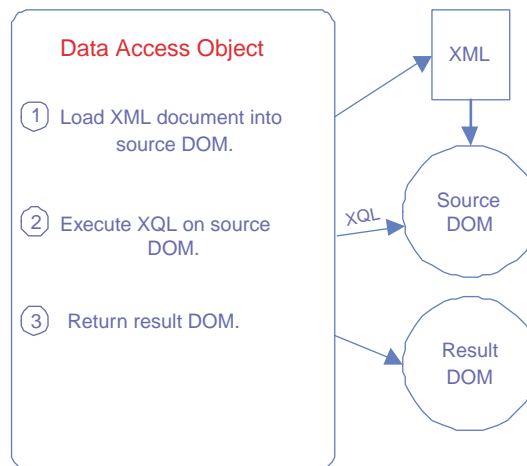


Figure 3.5
Data access object
processing using XQL

In the XQL version of our object, we create two instances of `org.w3c.dom.Document`. One will refer to the XQL data source document and the other to the result set document.

```
// DOM for the source document
Document srcDoc = DOMUtil.createDocument();
```

```
// DOM for the output document
Document rsltDoc = DOMUtil.createDocument();
```

Then we load the source document.

```
DOMUtil.parseXML(
    new FileInputStream(fileName),
    srcDoc,
    false, // Parse mode: nonvalidating
    DOMUtil.SKIP_IGNOREABLE_WHITESPACE
);
```

Next, we create an XQL query string and execute it, creating the result document.

```
String query = "//customer[@id='" + customerId + "']";
XQL.execute(query, srcDoc, rsltDoc);
```

Finally, we convert the result document into a JDOM document and return it to the caller.

```
org.jdom.input.DOMBuilder builder
    = new org.jdom.input.DOMBuilder();
return builder.build(rsltDoc);
```

These steps represent the interesting code in our Customer data access object using XQL. The full code for this class is contained in listing 3.6. This implementation uses a Java XQL implementation from the German National Research Center for Technology (GMD). Note that if we chose not to expose a JDOM Document interface, the CustomerDAOX object could simply return the org.w3c.dom.Document reference instead. This implementation returns a JDOM Document so it will work with the CustomerDataBean session EJB from the earlier example.

Listing 3.6 The Customer data access object using XQL

```
import de.gmd.ipsi.xql.*;
import de.gmd.ipsi.domutil.*;
import org.w3c.dom.*;

import java.io.FileInputStream;

/**
 * A data access object
 * for customer data
 * using XQL
 */
public class CustomerDAOX {

    protected String fileName = null;
```

```

public CustomerDAOX(String fileName) {
    this.fileName = fileName;
}

/** Return customer data as a JDOM Document */
public org.jdom.Document getCustomerInfo(String customerId)
    throws CustomerNotFoundException {

    Document srcDoc
        = DOMUtil.createDocument();

    Document rsltDoc
        = DOMUtil.createDocument();

    try {

        DOMUtil.parseXML(
            new FileInputStream(fileName),
            srcDoc,
            false, // Parse mode: non-validating
            DOMUtil.SKIP_IGNOREABLE_WHITESPACE
        );
    } catch (Exception e) {
        throw new CustomerNotFoundException(customerId, e);
    }

    String query = "//customer[@id='" + customerId + "']";
    XQL.execute(query, srcDoc, rsltDoc);

    org.jdom.input.DOMBuilder builder
        = new org.jdom.input.DOMBuilder();
    return builder.build(rsltDoc);
}

// other methods here to create and update customers
}

```

Parses
org.w3c.dom.Document
from file

Executes XQL

Returns resulting
JDOM Document

The code in listing 3.6 can be combined with the code in listing 3.2 to yield a robust data access object that can support both relational and XML data sources.

3.2.2 Storing XML data

Querying XML data is only useful if XML data repositories exist. The low-tech form of XML repository building is simply using a file system and XML files. This works well for small applications that can tolerate the overhead and performance characteristics of managing a group of XML-based text files. For larger applications and those that do not wish to manage a repository themselves, a more enterprise-ready solution is required. Throughout this section,

we examine relational databases, XML databases, and PDOM as storage options for your application.

Using relational databases

Virtually all the major players in the relational database world now offer XML integration capabilities in their database management systems. These vendors include Oracle, IBM, and Microsoft. The level of XML support varies by vendor, as does the mechanisms by which your XML data is converted to and from the relational data model. Therefore, should you use an RDBMS to store your XML data, your data-aware object implementations are likely to be closely tied to your chosen vendor. Also, be sure to thoroughly load-test the data layer of such applications, because the XML to relational transformation process is being done by the underlying database management system.

If you are uncomfortable using a proprietary mechanism to store XML in a relational database, or if your database does not support XML, you can always convert the data yourself. This gives you total control of the mapping between your XML and relational data models, at the expense of additional development and testing time. In section 3.1, we wrote a data access object that translated customer data between a JDOM document and a database table. We noted that the conversion process was too specific to customer data to be useful in other situations.

A generic data access object

To create a more generic version of the data access object, we need to change the `getCustomerData` method to accept an SQL query string rather than use its own, hard-coded one.

```
public Document getData(String SQL) throws Exception {
```

We then execute the query and obtain the JDBC `ResultSetMetaData` object to inspect query results.

```
    ResultSetMetaData rsmd = rs.getMetaData();  
    int cols = rsmd.getColumnCount();
```

We can then build the JDOM document using the information in the `ResultSetMetaData`. This document will generically represent a `result-set`, with some number of `row` nodes. Each `row` will have one `column` data node for each column in the result set. Additionally, we can add attributes to the `column` nodes to tag each with a column name and Java data type.

```
    while (rs.next()) {  
        Element row = new Element("row");
```

```

row.addAttribute("row-number", String.valueOf(i));
for (int j = 1; j <= cols; j++) {
    Element column = new Element("column");
    column.addAttribute("column-name", rsmd.getColumn(j));
    column.addAttribute("column-type", rsmd.getColumnTypeName(j));
    column.addContent(rs.getString(j));
    row.addContent(column);
}
root.addContent(row);

```

Listing 3.7 shows our more generic implementation of the relation to XML conversion process.

Listing 3.7 The generic JDBC data access object

```

import org.jdom.Document;
import org.jdom.Element;
import javax.sql.DataSource;
import java.sql.*;

public class GenericDAO {

    protected DataSource ds = null;

    public GenericDAO(DataSource ds) {
        this.ds = ds;
    }

    public Document getData(String SQL)
        throws Exception {
        Document doc
            = new Document(new Element("Result-Set"));
        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            con = ds.getConnection();
            stmt = con.createStatement();
            rs = stmt.executeQuery(SQL);
            ResultSetMetaData rsmd = rs.getMetaData();
            int cols = rsmd.getColumnCount();
            int i = 1;
            Element root = doc.getRootElement();
            while (rs.next()) {
                Element row = new Element("row");
                row.addAttribute("row-number", String.valueOf(i));
                for (int j = 1; j <= cols; j++) {
                    Element column = new Element("column");
                    column.addAttribute("column-name",
                        rsmd.getColumn(j));
                    column.addAttribute("column-type",
                        rsmd.getColumnTypeName(j));

```

Inspects metadata

1

Iterates and
populates
JDOM


```

        column.addContent(rs.getString(j));
        row.addContent(column);
    }
    root.addContent(row);
    i++;
}
} catch (Exception e) {
    throw e;
} finally {
    if (rs != null)
        try { rs.close(); } catch (SQLException sqle1) {}
    if (stmt != null)
        try { stmt.close(); } catch (SQLException sqle2) {}
    if (con != null)
        try { con.close(); } catch (SQLException sqle3) {}
}
// return a JDOM Document
return doc;
}

// other methods here to insert and update data
}

```

Listing 3.8 highlights the changes to the `CustomerDataBean` required to use the new, generic data access object.

Listing 3.8 CustomerDataBean using the generic DAO

```

import javax.sql.DataSource;
import javax.ejb.EJBException;
import org.jdom.Document;

public class CustomerDataBean2
    implements javax.ejb.SessionBean {
    public javax.ejb.SessionContext ctx;

    // transient so it won't be
    // serialized on passivation
    public transient GenericDAO gDAO;

    public void ejbCreate() {
        buildDAO();
    }

    public org.jdom.Document
        getCustomerInfo(String customerId) {
        String SQL = "select * from customers where " +
            "customerId='" + customerId + "'";
        Document custData = null;
    }
}

```

A session bean that retrieves customer data as a JDOM Document using the generic DAO

Takes a customerId and gets that customer's information using the generic DAO

```

try {
    custData = gDAO.getData(SQL);
} catch (Exception e) {
    throw new
        EJBException("Error while retrieving customer data.", e);
}

// we could translate the "Result-Set" XML structure
// to the original customer format here if we
// needed to

return custData;
}

protected void buildDAO() throws EJBException {
    try{
        javax.naming.Context jndiCtx
            = new javax.naming.InitialContext();
        javax.sql.DataSource ds =
            (javax.sql.DataSource)
                jndiCtx.lookup("java:comp/env/jdbc/CustomerDB");
        gDAO = new GenericDAO(ds);
    } catch (Exception e) {
        throw new EJBException(e);
    }
}

public void ejbRemove() { }

// restore Data Access Object when activated
public void ejbActivate() { buildDAO(); }

public void ejbPassivate() { }

public void setSessionContext(javax.ejb.SessionContext ctx) {
    this.ctx = ctx;
}
}

```

Uses DAO to get customer data

The buildDAO method looks up the data source in the environment and passes it to the DAO's constructor

To jump-start your data access object code development efforts, you can also use an object-relational (O/R) mapping product such as WebGain TopLink. While these products won't generate generic data access objects, they will write the code for doing specific relational data operations. You can modify the code generated by these tools to include the necessary XML transformations. Given time constraints, this may be the most practical approach if your data access and translation process cannot be generalized.

Using XML databases

An alternative approach for data storage is to use an XML database. One of the most notable XML database systems at the time of this writing is Software AG's Tamino product. Tamino is an e-business application suite, driven in large part by its XML data server. This data server stores XML in its native, hierarchical format, meaning that no translation between data models is needed. Also, Tamino and products like it will be quick to implement XQuery functionality as the standard solidifies.

Before rushing out to purchase an XML database product, realize that XML data servers are relatively new technology and have not been road tested in production environments like RDBMSs have. As a seasoned architect, you must always make technology decisions based on a balance between the potential benefits of a new technology and the risks associated with using it.

PDOM

If you decide to manage your own XML data, your attention will soon turn to maximizing the performance of the XML data store. XML data sets are trees that can be expensive to iterate over and search. An option to consider is using the PDOM API. PDOM, the Persistent Document Object Model, was designed to enhance the performance of the GMD's XQL implementation we discussed in the previous section. It stores XML in an indexed, binary file and uses a memory-paging algorithm to speed operations on large XML documents. Using a PDOM instead of a DOM is simple, because the API exposes an implementation of the `org.w3c.dom.Document` interface.

Listing 3.9 contains a data access object that uses a PDOM file instead of an XML text file. This example is very similar to Listing 3.6, but uses a PDOM `PDocument` as the data source instead.

```
PDocument srcDoc = new PDocument(fileName);
```

The XQL and JDOM building process remain the same, as you see. The PDOM API also provides the ability to commit changes made to the DOM in memory back to the binary file. This can be useful in a transactional context. Detailed information on PDOM can be found at the GMD's XQL web site, <http://xml.darmstadt.gmd.de/xql/>.

Listing 3.9 Using the PDOM API

```
import de.gmd.ipsi.pdom.*;
import de.gmd.ipsi.xql.*;
import de.gmd.ipsi.domutil.*;
import org.w3c.dom.*;
```

```

import java.io.IOException;
public class CustomerDAOP {
    protected String fileName = null;
    public CustomerDAOP(String fileName) {
        this.fileName = fileName;
    }
    /** Return customer data as a JDOM Document */
    public org.jdom.Document
        getCustomerInfo(String customerId)
            throws IOException {
        // DOM for the source document
        // PDocument implements org.w3c.dom.Document
        PDocument srcDoc = new PDocument(fileName);
        // DOM for the output document
        Document rsltDoc = DOMUtil.createDocument();
        String query = "//customer[@id='" + customerId + "']";
        XQL.execute(query, srcDoc, rsltDoc);
        org.jdom.input.DOMBuilder builder
            = new org.jdom.input.DOMBuilder();
        return builder.build(rsltDoc);
    }
    // other methods here to create and update customers
}

```

A data access object for customer data using PDOM and XQL

← Path to PDOM file

← Returns JDOM document

← Executes XQL

Table 3.3 summarizes the current solutions for storing XML data.

Table 3.3 XML repository solutions

Solutions	Pros	Cons
File system and XML files	Human readable	Must reparse files on each access. Number of files becomes unwieldy.
RDBMS	Enables you to stay with your current vendor product	Requires use of proprietary storage mechanism.
XML database	Storage in native XML format	Unproven technology
PDOM	Files are already parsed	New technology, relies on file system storage.

3.2.3 *When not to use XML persistence*

There are many circumstances in which it is not appropriate to store your data as XML, even if your application is internally XML-based. One key example is transactional data, especially when it must be highly available and replicated. Relational database systems are very good at optimizing queries, ensuring data integrity, and replicating data across instances. There will be a time when XML databases can claim the same performance and reliability characteristics, but not yet.

Another key situation in which you should still prefer relational databases is when your data are highly interrelated. Using an RDBMS will allow you to normalize those data, eliminating redundancies and increasing integrity. XML technologies such as XLink and XPointer are not nearly mature enough at this point to allow you to normalize an XML data repository. Deriving useful data from disparate XML data sets is currently difficult and expensive in terms of resource utilization and processing time.

As XQuery and other XML technologies are implemented and refined over time, these concerns will diminish. In the nearer future, storing your data relationally and converting to XML is still your best option in many situations.

3.3 *Summary*

In this chapter, we examined the ways XML might be used to add flexibility and generality to your internal application components. XML can be used in component interfaces, as an internal data representation format, and as a persistent storage format. The JDOM API provides an easy method of working with XML DOMs as data value objects instead of using proprietary software objects. Using generic XML data structures allows your application to manipulate and transform data in generic ways. It also makes your application component interfaces generic and enhances reusability.

XML technologies for storing and retrieving persistent XML data are in their infancy. For now, converting between XML and relational formats is required in most production systems. Over time, tools such as XQuery and XML database products plan to eliminate this requirement for many types of applications. For now, implementing the data access object and doing some form of conversion between data formats is a useful approach.

Using XML throughout your application may not be appropriate in situations where response time must be minimized or system memory is severely constrained. Using XML is more resource-intensive than using a more specific approach. As the technology matures, this will become less of an issue. For now, carefully consider the costs and benefits of using XML throughout your application before proceeding to design it.



Application integration

This chapter

- Introduces system integration concepts
- Suggests techniques for successful J2EE
- Systems integration
- Demonstrates implementation of J2EE-based web services
- Uses JAX APIs to enable web services

This chapter is about using XML technology to integrate your J2EE application with other applications and services. We describe the systems integration activity and its central role in the success of your distributed system. We also present a proven approach for integrating independent systems, including the major architectural patterns used to do so. Traditional approaches to systems integration are compared and contrasted with XML-based ones.

The remainder of the chapter focuses on the web services architecture that many believe is the future of distributed systems on the Internet. We define and discuss SOAP and its role in web services. We then provide examples of producing, locating, and consuming web services from within the J2EE environment.

4.1 *Integrating J2EE applications*

In a distributed environment, connecting your application to other applications, services, and data sources is essential. How these connections are made can impact the performance, reliability, and functionality of your application more than its own internal design. Synchronous interaction with other systems over a network can have a dramatic effect on the overall performance of your application. System interaction also means ensuring data integrity between applications, which can be complex. However, a standalone application that does not have access to other enterprise systems cannot do much.

The work of gluing together independent applications, services, and data stores is referred to as *systems integration*. This is a discipline of its own that has become as important in recent years as distributed application development itself. In this section, we explore some systems integration possibilities in the J2EE context. We identify the general architectural patterns for system integration and examine the ways XML can enhance that integration.

4.1.1 *Traditional approaches to systems integration*

Perhaps the most difficult part of integrating your application into its environment is determining where to start. In the majority of cases, you are developing a new application and need to integrate it with existing systems. Each of these systems provides a limited set of options for interacting with it. Some systems expose remote APIs or object interfaces. Others support only asynchronous messaging. Some may not offer any integration options at the application layer, but can be fooled into integration at the data layer.

Although specifics vary from system to system, your integration options can be broadly classified into four architectural patterns. These patterns vary in levels of complexity from simple data integration to tightly coupled software

objects. Each has a distinct set of advantages and disadvantages, and the extent to which XML can play a role varies by pattern as well. The relative complexities and sophistication of these patterns are shown in figure 4.1. These patterns should look familiar to you. First, we will briefly review the traditional approaches to provide you with some context. Then, in section 4.1.2, we will examine the role that XML plays in each of these integration techniques.

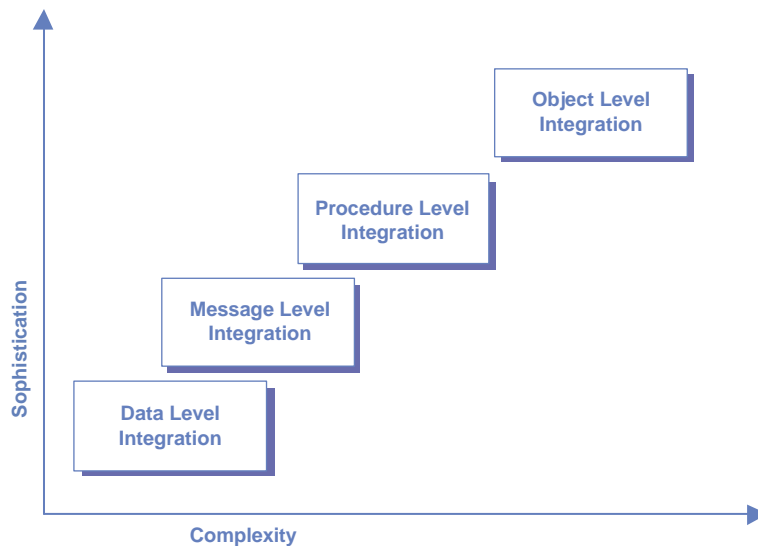


Figure 4.1 Relative complexity of systems integration patterns

Data level integration

This is the simplest architectural pattern for systems integration and may be the quickest to implement. To integrate at the data layer, your application shares access to an enterprise data store with one or more other applications.

For example, your application might query a remote data store to retrieve order history information for a particular customer. The data store itself belongs to another application, but your application has read-only access directly to the data.

In another situation, your application might update a remote data store. If, for example, your application needs to update customer information in another system, it might simply execute an SQL statement against the other system's database. Data level integration is depicted in figure 4.2.

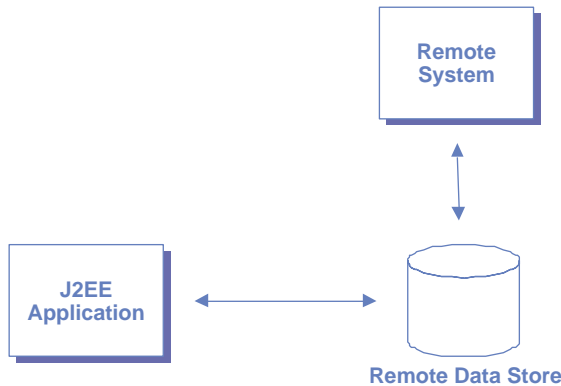


Figure 4.2
Integrating your application
at the data level

Advantages of data level integration include the following:

- This integration type can be simple. When done properly, the remote system has no knowledge of your application and operates without modification.
- This pattern can work with closed systems that do not expose any other API for integration.

Disadvantages of data level integration include the following:

- Management overhead can be complex when updating data owned by a remote system. Your application is responsible for maintaining data integrity between its own data and the remote system's data. This may involve reproducing some of the remote system's application logic in your integration components.
- Mapping your application data model to that of the remote system may be complex. Developing the translation mechanism may be difficult. Also, if the integration is synchronous, the performance cost of translating and transferring data may be unacceptable.
- Depending on your network and security configuration, you may not be able to reach the remote data store directly. In those situations, this pattern can be combined with message level integration, which we discuss next.

To implement this pattern in J2EE, you can use a data access object to wrap remote database interaction and decouple it from an EJB. If the remote data being accessed is client-specific, the DAO could be invoked from a session bean. If the data is shared, the DAO could be used as a helper for an entity bean.

Message level integration

Message level integration is the most flexible and best performing of the system integration patterns. In this pattern, your application packages data into a request and transmits it over the network to the remote system asynchronously. The remote system unpackages the data and performs processing. This is depicted in figure 4.3. Because the interaction is asynchronous, your application does not wait for the remote system to process the message, resulting in better overall performance.

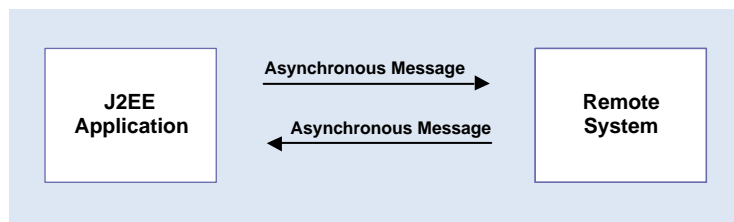


Figure 4.3 Message level integration to a remote system

In the reverse case, a remote system can generate messages and send them asynchronously to your application. These messages might indicate the result of some requested processing or initiate some new processing within your application.

Advantages of message level integration include the following:

- Message level integration provides loose coupling between your application and remote systems. Message creation and message processing are completely separate from one another, and may occur at different times. Your application and the remote system only need to agree on the format and meaning of the messages.
- Message level integration maximizes performance. Your application does not have to wait for remote processing to complete before continuing on.
- Message level integration is scalable. Using message-oriented middleware (such as a JMS provider), messages can be rerouted on the fly as resources dictate. Also, the messaging middleware can provide guaranteed delivery of messages, removing some failure-handling burden from your integration components.

- Message level integration is flexible. The implementation code that generates the message can be modified without altering the code that processes the message, and vice versa.

Disadvantages of message level integration include the following:

- Message level integration only supports the asynchronous interaction model, by definition.
- This pattern may require adapter code to be written at the remote system. Some systems natively support a set of known message types and a delivery mechanism that can be extended to accommodate your requirements. In other situations, you may have to write code that receives messages on behalf of the remote system and interacts with it in some other way.

JMS makes implementing message level integration easy. To implement outbound messaging, you simply publish a message to a JMS topic or queue. This can be done from within an EJB or a dependent object. Consider using the Business Delegate pattern from chapter 2 to separate the messaging code from your EJB. Inbound messaging is even easier. Simply implement a Message Driven EJB to process messages arriving on a specific JMS queue.

Procedure level integration

This pattern is an extension of message level integration that supports synchronous interaction between systems. This is nothing more than a coordinated pair of messages, a client/server request and reply. Many enterprise systems support this pattern, referring to it as either remote procedure calls (RPC) or remote function calls (RFC). This interaction type is depicted in figure 4.4.

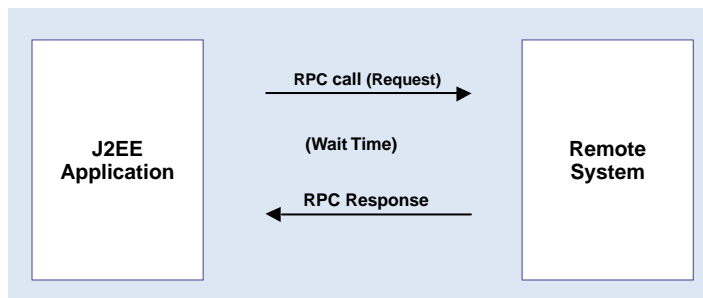


Figure 4.4 Procedure level integration

Advantages of procedure level integration include the following:

- Procedure level integration supports synchronous interaction as an extension of the message level integration model. This is often an essential requirement in real-time distributed systems.

Disadvantages of procedure level integration include the following:

- Procedure level integration slows application performance, since your application must wait for remote processing to complete before continuing.
- Procedure level integration adds a great deal of complexity to the failure model between systems. Your application must determine how remote failures are detected and handled, perhaps including logic to timeout requests and retry them.

Procedure level integration can be complex, and therefore should be encapsulated by a Business Delegate or other dependent object. If remote system resources are constrained, you should also consider pooling a fixed number of delegate objects within each EJB container being used. In this way, you can predict the maximum number of concurrent connections being made to the remote system. The delegate object might also contain logic to rollback its remote changes when used in a transactional EJB context. Connection details, object pool size, and other configuration parameters should be kept in the deployment descriptor of the EJB using the delegate.

Object level integration

Object level integration is a specialized integration pattern. It can be used between systems that support a common distributed object architecture, such as CORBA or DCOM. In this pattern, some well-known object acts as an intermediary between client and server software object. This intermediary (the ORB in CORBA) provides a set of services that clients can use to locate a desired server and invoke it. The general case for object integration is depicted in figure 4.5. Once the client has located the service, the server object provides a remote stub of itself to the client that it can use locally. The stub code translates local method calls into remote ones. On the server side, there is a client skeleton providing the same proxy functionality to the server object. This is depicted in figure 4.6.

Advantages of object level integration include the following:

- This type of interaction permits object-oriented access to remote applications. This is most useful when the application itself is distributed, since the object models of independent systems may not be completely compatible.

- Marshalling (packaging) of data and transmission are managed by the distributed object system.
- Remote failures can be detected via code level exception handling.

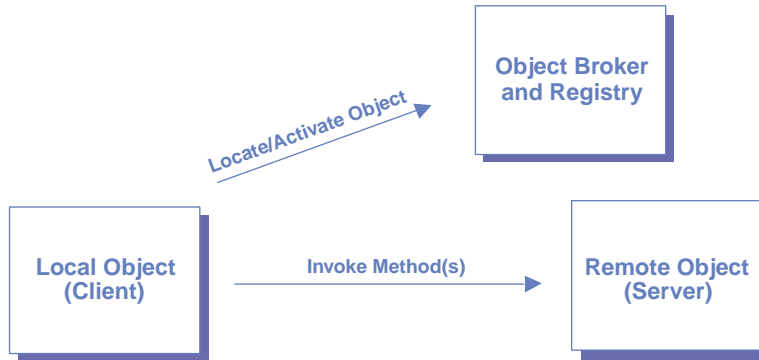


Figure 4.5 Object level integration

Disadvantages of object level integration include the following:

- This type of integration tightly couples your application components with those of the remote system. The two systems become integrated at the code level via remote interfaces.
- Both your application and the remote system must support the same remote object architecture, directly or through a bridging tool like Intrinsyc Software's JIntegra product. For many legacy systems, this type of integration is not possible due to lack of support for distributed objects.
- Using an intermediary lookup service remotely takes time and can degrade overall system performance if references to remote objects cannot be effectively cached and reused.

EJB supports object level integration extensively via RMI. In fact, this integration model is the basis of all interactions between EJBs and their clients, both remote

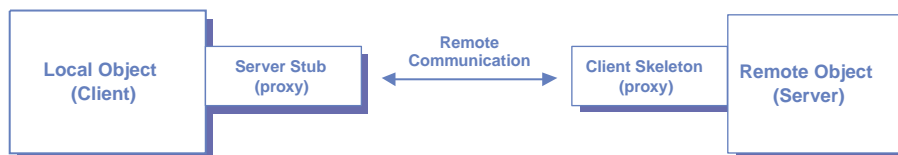


Figure 4.6 Remote object interaction using proxies

and local. Since RMI can be done over IIOP, EJB components can participate in CORBA systems. Additionally, products such as JIntegra can be used to access EJB from DCOM systems. Consider using the Session Façade or Aggregate Entity pattern to encapsulate the integration points between your internal components and remote systems.

Constraint-based modeling

Now that we have identified our system integration options, we need a method for choosing which to use in a given situation. An effective approach is to begin your analysis without assumptions and narrow your options by considering the limitations and capabilities of the systems to which you must connect. This is known as *constraint-based modeling*, in which the existing environment guides much of the integration architecture.

Before walking through an example, note that flexibility and performance are two of the key design goals from chapter 1. Since message level integration is the fastest and most flexible type of integration, we suggest you use it whenever possible. This would make your life much easier if it were not for synchronous interaction requirements. If your application must obtain or update remote data in real-time for whatever reason, message level integration goes out the window.

To demonstrate how constraint-based modeling works, let us consider the following example. Say we are building an e-commerce application in J2EE and have a requirement to obtain the most current pricing for each customer from our enterprise ERP system whenever a customer begins to make a purchase. Furthermore, we know that the ERP system does not store prices statically for each customer, but performs a complex calculation to arrive at a unit price for each product. This procedure requires data that is otherwise irrelevant to our e-commerce application, and reproducing it on our side is infeasible.

This requirement adds a constraint to our integration model; interaction must be synchronous. This eliminates message level interaction and leaves us with data, procedure, and object level options. Now we inspect the documentation for the ERP system and note that it exposes a set of RPCs that can be used to obtain customer pricing, but does not support CORBA or any other object architecture. We are now down to procedure and data level integration options. Data level integration is infeasible, because it is the pricing algorithm itself to which we need access in the ERP system, not the data upon which it operates.

This leads us to the conclusion that this integration point will use procedure level integration via the existing RPC mechanism exposed by the ERP

system. Note that we did not really make any decisions in this simple example. The requirement and capabilities of the remote system mandated the decision for us. In more complex examples, you may need to choose between two viable options. If, for example, the ERP system did support object level integration, you would need to make a decision. Since procedure level integration involves less coupling of the systems, you might choose to stick with the RPC option. The point being that you may have a choice and need to apply judgment after the constraint-based modeling exercise.

4.1.2 *XML-based systems integration*

XML is arguably the most important development in systems integration in the past twenty years. The ability to generically describe, manipulate, and transform data is allowing all types of enterprise systems to be connected in ways never before possible. In this section, we examine the ways XML can enhance J2EE systems integration in each of the four patterns described in section 4.1.1. As you will see, XML technology provides most of its value at the message and procedure level, but can be used in specialized ways at the data and object integration levels as well.

XML and data level integration

At the data level, XML provides a standard mechanism for mapping data models between your J2EE/XML application and the data store of a remote system. An XSLT engine can restructure XML data into any arbitrary format, making your data translation work much simpler. In the near future, XQuery will enhance this capability greatly through its ability to group, join, derive, and sort XML data. While the work of translating your XML data into remote database calls is still your responsibility, XSLT and XQuery are available to assist you in certain situations.

This method is clearly advantageous compared to data level integration without XML. Tools such as XSLT and XQuery allow your application to manipulate the data coming from the remote data store. This means that your application can be flexible and easily adapt to changes in the data format.

For example, in a simple case you may be reading data from a remote repository of XML flat files. When your application retrieves one or more files, it performs an XSLT transformation to put the data in a format that is meaningful to your application. This transformation only requires an XSL stylesheet that can easily be modified if the source or target output format changes. XML and data level integration are illustrated in figure 4.7.

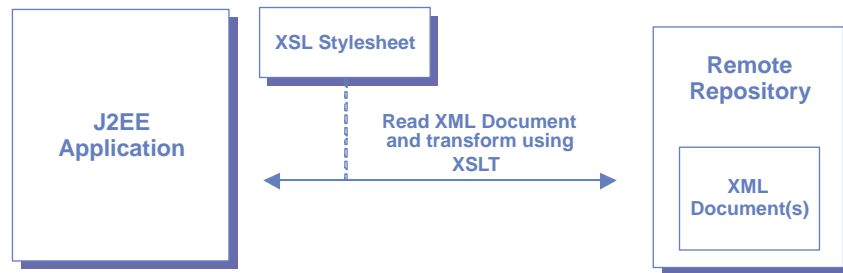


Figure 4.7 Using XML for data level integration

XML and message level integration

As noted in chapter 2, much work has been done in the area of XML messaging. Leveraging the ability of XML to describe, validate, and structure messages between systems, many industry groups have developed standards for XML messaging to facilitate B2B commerce and electronic document interchange (EDI). These standards include RosettaNet, cXML, and ebXML. While each of these efforts began under different circumstances, they are attempting to do very similar things:

- Define an unambiguous XML language for the exchange of business data
- Define a standard mechanism for integrating new business partners into existing processes
- Define a standard mechanism for transmitting XML documents between organizations
- Define a security model for XML data interchange

Fortunately, these numerous standards are moving in the direction of unification, particularly in the areas of data transport and partner integration. The Simple Object Access Protocol (SOAP) is quickly gaining acceptance as a standard mechanism for transmitting XML messages, both synchronously and asynchronously. We introduced SOAP in chapter 2, and discuss it in detail in section 4.3. In the area of new partner integration, many XML standards bodies are now supporting the Universal Description, Discovery, and Integration Interface (UDDI), which we discussed briefly in chapter 3 as well. We discuss UDDI implementation details further in section 4.3 as part of our discussion of web services.

The Java community is also hard at work defining standard APIs for using XML-based messaging protocols like SOAP. The most promising proposal at the message integration level at the time of this writing is the Java API for XML Messaging (JAXM). JAXM provides an API for sending and receiving SOAP messages. We demonstrate the use of JAXM in section 4.3.

XML and procedure level integration

In section 4.1.1, we noted that procedure level integration is really just a coordinated pair of asynchronous messages exchanged between systems. As such, the same XML technologies used in message level integration apply at this level too. SOAP can be used in two-way (synchronous request-response) messaging, due in part to its explicit binding to the HTTP protocol. Additionally, the JAXM API can be used in either one-way or two-way mode. The examples in section 4.3 demonstrate this.

There is also ongoing work to develop a Java API for XML-RPC, called JAX-RPC. This API will be closely aligned with SOAP and other XML messaging technologies championed by the W3C. The JAX-RPC effort is focused on defining a mapping between Java classes, interfaces, and data types and XML data types and messaging protocols. This mapping is intended to make RPC over XML protocols like SOAP easier to use in the Java environment.

Since JAX-RPC is in its very early stages, it is not possible to demonstrate its use in this book. It is possible that the entire API will be subsumed into another API such as JAXM in the future. However, you should watch the development of this API closely going forward.

XML and object level integration

XML might also be used in special cases where object level interaction is required. Consider the situation in which you need to exchange a shared object with another system, but using a distributed object architecture is not possible. This could be due to security concerns or other technical limitations. In such a situation, you might decide to serialize a Java object as an XML document, transmit it to the remote system, and reactivate it there. As long as the remote system is Java-based and the object is self-contained, this can be accomplished using JAXM (over SOAP) and the Java API for XML Binding (JAXB). This is an admittedly contrived example, and is intended only to open your mind to the possibility of using XML even in situations of tightly coupled, all-Java systems integration efforts too.

You might also use XML to encode the data values passed back and forth between remote objects. You could, for example, serialize a DOM structure to

a string and use it as a parameter in a remote method invocation. Serializing a Java value object to XML and passing it to a non-Java based remote object can be an easy way to convert data objects between platforms.

4.2 *A web services scenario*

In section 4.1, we reviewed several different methods for integrating your applications with remote systems. Due to the newness of SOAP and web services, and the great potential that they have to make a significant impact on application integration, we review and analyze them in detail in sections 4.3 and 4.4.

In order to bring some context to the discussion in the next two sections, we apply a real world example involving a manufacturer and a distributor. The manufacturer makes products that the distributor sells. There has been a slight problem with this arrangement because the manufacturer changes product information such as pricing every so often, and the distributor is still working with the old data. Both companies decided to analyze the use of SOAP and web services to transmit product information between their companies. We will develop the code necessary to make this integration happen over the next two sections. Figure 4.8 depicts the communication between both companies.



Figure 4.8 Manufacturer and distributor application integration

4.3 *J2EE and SOAP*

Before we apply SOAP as a solution to our problem, we must understand what SOAP actually is. SOAP is a lightweight mechanism for exchanging messages in a distributed environment. The SOAP specification is currently at version 1.1, although version 1.2 is already a working draft. Detailed information about SOAP can be found at <http://www.w3.org/2000/xp/>. The SOAP specification addresses four main areas, as shown in table 4.1.

Table 4.1 SOAP specification components

SOAP construct	Description
Message envelope	The message envelope describes what a message is and who should consume it.
Encoding rules	SOAP defines encoding rules for serializing application-defined data types.
RPC support	SOAP defines constructs to support RPC interaction between message senders and receivers. SOAP also supports asynchronous (one-way) messages.
HTTP binding	A SOAP message may be bound to the HTTP protocol and wrapped in an HTTP packet.

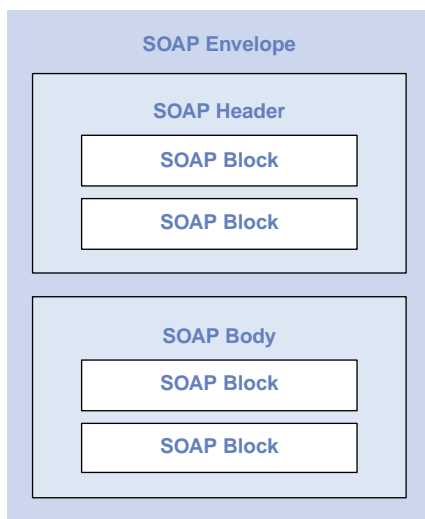


Figure 4.9 SOAP message structure

The structure of a basic SOAP message is depicted in figure 4.9. The optional SOAP header element may contain meta-information specific to a SOAP message itself, including any special identifiers needed by participating applications. The SOAP body contains the payload of the message, which can be an application specific request, response, or an asynchronous message. Both the header and the body can be further structured into SOAP blocks. For example, a SOAP body could contain two blocks, each containing a separate remote procedure call.

4.3.1 *Creating a simple SOAP message*

Now that we know something about what SOAP is, let us see what it looks like. To do so we will create and send a SOAP message using our example scenario from section 4.2. Let us suppose that our distributor wants to know the price of a product with a product identifier of 123456. The distributor's system makes an RPC method call to the manufacturer using a SOAP message. The distributor calls a SOAP method named `GetProductPrice` and passes a



Figure 4.10 Simple RPC call using a SOAP message

`productId` parameter. The manufacturer returns a price of \$99.95 to the distributor. Figure 4.10 illustrates this chain of events.

SOAP request and response packets

Before we look at the Java code necessary to facilitate this interaction, we examine the SOAP messages that are transmitted between the two companies. Listing 4.1 shows the SOAP request and response packets for the `GetProductPrice` method call. As you can see, the request packet provides a product identifier parameter and the response packet contains the current price for the specified product.

Listing 4.1 SOAP request and response packets

```

<!-- SOAP request packet -->
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetProductPrice xmlns:m="MyCompany-URI">
      <productId>123456</productId>
    </m:GetProductPrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
-----
<!-- SOAP response packet -->
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>

```

← Parameter passed to remote method

```

    <m:GetProductPriceResponse xmlns:m="MyCompany-URI">
      <Price>99.95</Price>
    </m:GetProductPriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

← Response parameter

The SOAP request and response packets must be transmitted over a network between our manufacturer and distributor. We discuss the transport mechanism for SOAP messages in the next section.

SOAP message transport

How does the SOAP message get transported? Currently, HTTP is the only transport protocol explicitly defined for SOAP. When using HTTP, SOAP messages are wrapped in either an HTTP request or response packet. Listing 4.2 shows the messages from listing 4.1 being transmitted via HTTP.

Listing 4.2 SOAP messaging over HTTP

```

POST /ProductPriceQuote HTTP/1.1
Host: www.supplierserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "MyAction-URI"

```

HTTP request headers

```

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetProductPrice xmlns:m="MyCompany-URI">
      <productId>123456</productId>
    </m:GetProductPrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

```

HTTP response headers

```

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetProductPriceResponse xmlns:m="MyCompany-URI">
      <Price>99.95</Price>
    </m:GetProductPriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

```
</m:GetProductPriceResponse>  
</SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

4.3.2 Using SOAP with Attachments

The data transmitted between companies is not always textual in nature. Graphics files and PDF documents are two very common examples of data that companies often share. In section 4.3.1, we sent a textual parameter in the request, a product id, and received a textual response. But what mechanism can we use to transmit binary data using SOAP?

There is an extension to the basic SOAP message structure that can accommodate non-SOAP (and non-XML) attachments to SOAP messages. This extended specification, SOAP 1.1 with Attachments, defines a mechanism for building multipart SOAP messages based on MIME encoding. This standard uses the same encoding mechanism used in Internet email systems that allows files to be attached to messages.

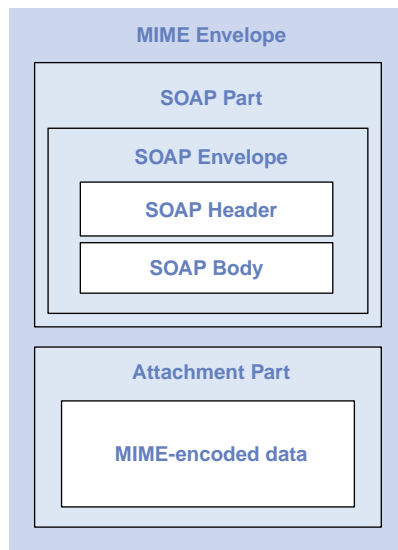


Figure 4.11 SOAP with Attachments message structure

The structure of a SOAP 1.1 with Attachments message is depicted in figure 4.11.

A complex SOAP message is divided into Parts, which can be SOAP Parts or Attachment Parts. SOAP Parts contain a SOAP Envelope, while Attachment Parts can contain any type of MIME encoded data. This extension is very useful for sending binary data (images, etc.) along with SOAP messages. The SOAP 1.1 with Attachments specification can be found at <http://www.w3.org/TR/SOAP-attachments>.

A SOAP with Attachments example

In order to examine a SOAP message that contains an attachment, we add another message to our manufacturer and distributor example. In this case, the manufacturer updates a product's catalog image, which causes a SOAP message to be sent to the distributor. The message includes the new image as a binary attachment.

Listing 4.3 demonstrates the HTTP interaction for an asynchronous SOAP message with an attachment (the product image) being sent from our manufacturer to our distributor. This example includes an updated image for the product with an id of 123456.

Listing 4.3 HTTP SOAP Message with an Attachment

```

POST /imageUpdate HTTP/1.1
Host: www.mycompany.com
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
    start="<productImageUpdate.xml@mycompany.com>"
Content-Length: XXXX
SOAPAction: http://schemas.mycompany.com/Product-Images
Content-Description: An updated thumbnail image for product 123456

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <productImageUpdate.xml@mycompany.com>

<?xml version='1.0' ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<myco:productImage_id="123456" type=thumbnail
    xmlns:myco="http://schemas.mycompany.com/Product-Images">

    <imageData
        href="cid:product123456thumb.jpg@mycompany.com"/>
</myco:productImage>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <product123456thumb.jpg@mycompany.com>

...Raw JPEG image..
--MIME_boundary--

```

HTTP headers

The cid attribute links the attached file contents to the SOAP message payload by its MIME block identifier

SOAP messages can also include digital signature information, so that you can verify the authenticity and validity of SOAP messages. This is based on the XML Signature recommended standard, also from the W3C. For more details, see <http://www.w3.org/TR/SOAP-dsig/>.

Getting more information about SOAP

A more detailed discussion of the SOAP protocol, encoding rules, and HTTP binding is beyond the scope of this book. We are primarily concerned with how to use SOAP from J2EE in this chapter. To learn more about what SOAP is and how it works in general, we refer you to the references on the subject that are cited in the bibliography.

4.3.3 Using JAXM for SOAP Messaging

In sections 4.3.2 and 4.3.3, we examined the contents of SOAP messages transmitted between our manufacturer and distributor. We have not yet discussed how these messages are sent between our systems. For the purposes of our discussion, we assume that the communication is taking place between J2EE applications. Do not forget, however, that one of the more popular features of SOAP is its ability to integrate heterogeneous systems (e.g., Microsoft and J2EE applications).

Currently, the only standard method for transmitting SOAP messages from Java is the Java API for XML Messaging (JAXM). JAXM, which we examined briefly in chapter 2, is currently being developed under the Java Community Process (JCP). JAXM is an API providing a standard mechanism for sending and receiving XML messages. The initial version of the JAXM specification supports SOAP 1.1 with Attachments, as described in the previous section.

Although it is currently in an early access release, JAXM will provide a familiar mechanism for using SOAP within your J2EE components. The API is similar to the Java Messaging Service (JMS) in structure, defining connection factories to abstract JAXM clients from the underlying SOAP provider. As you will see in the examples, JAXM is based on the JDOM API we have been using throughout the book.

Using a Business Delegate object for JAXM Messaging

In order to use JAXM, you must write Java code that performs its communication with a remote host using the JAXM API. In our example, we implement this code using the Business Delegate pattern. The Business Delegate pattern is used to simplify client access to an application service. The Business Delegate object hides the complexity of interacting with JAXM from the other components in our application. In our case, we will implement a session EJB as our Business Delegate object. Figure 4.12 depicts this scenario. For more information about the Business Delegate pattern, please see appendix A.

In listing 4.1, we examined the SOAP messages that were sent between our manufacturer and distributor when the `getProductPrice` method was called.

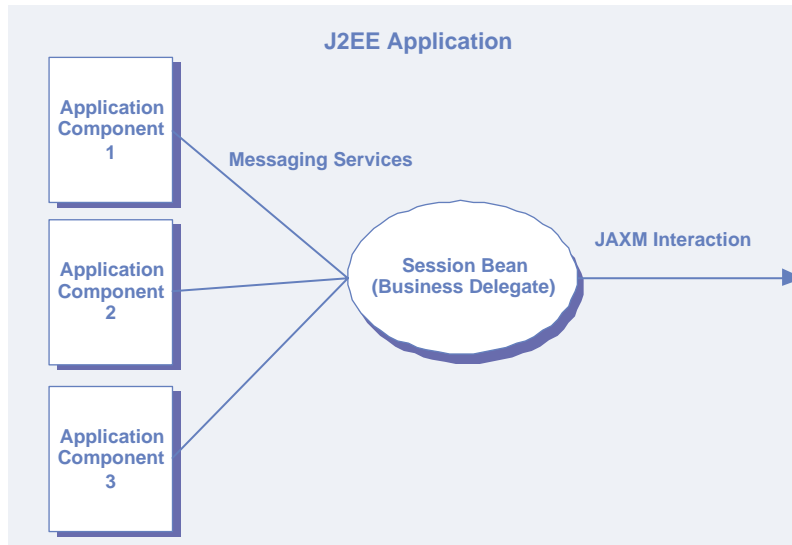


Figure 4.12 Using a Business Delegate object with JAXM

Now we examine the session EJB that generates the SOAP message and interacts with the remote host (manufacturer). The SOAP interaction is handled inside a Business Delegate's `getProductPrice` method, via JAXM. In this method, we use a JAXM connection to obtain a `MessageFactory` and create a JAXM Message.

```
SOAPMessageFactory mf = con.createMessageFactory();
SOAPMessage msg = mf.createMessage();
```

JAXM messages contain both a `SOAPPart` and zero or more `Attachment Parts`, as specified by the SOAP 1.1 with Attachments specification. Every JAXM message has a `SOAPPart` when it is initially created. Now that we have created the message, we must obtain a handle to the different parts of the message put some content in the message. We can obtain a handle to the envelope, header, and body of the message within the `SOAPPart`, as follows:

```
SOAPPart sp = msg.getSOAPPart();
SOAPEnvelope envelope = sp.getSOAPEnvelope();
SOAPHeader hdr = envelope.createSOAPHeader();
SOAPBody bdy = envelope.createSOAPBody();
```

After obtaining a handle to the `SOAPBody`, we can create a remote method call and bind its parameters.

```

SOAPBodyElement gpp
    = bdy.createSOAPBodyElement("GetProductPrice",
        Namespace.getNamespace("m", "MyCompany-URI"));

gpp.createSOAPElement("productId",
    Namespace.getNamespace("m", "MyCompany-URI"))
    .addContent(productId);

```

Finally, we bind the header and body to the envelope, create an `Endpoint` (destination address), and make the remote method call.

```

// Set the soap header and soap body on the envelope.
envelope.setSOAPHeader(hdr);
envelope.setSOAPBody(bdy);

// Create an endpoint for the recipient of the message.
Endpoint endPoint = new Endpoint("www.mysupplier.com");

// Get the price via RPC
SOAPMessage priceMsg = con.call(msg, endPoint);

```

After the call has been made, we can inspect the resulting message and retrieve the product price response.

```

String priceString
    = priceMsg.getSOAPPart().getSOAPEnvelope()
        .getSOAPBody().getChild("GetProductPriceResponse",
            Namespace.getNamespace("m", "MyCompany-URI"))
        .getTextTrim();

```

The full source for the Business Delegate is contained in listing 4.4. Once this class is implemented, any other component in our application that needs pricing information just needs to call its `getProductPrice` method.

Listing 4.4 A JAXM Business Delegate for SOAP RPC

```

import javax.xml.messaging.*;
import javax.xml.soap.*;
import org.jdom.*; ← JAXM is built on
                    top of JDOM
import java.net.*;
import java.io.*;
import java.util.*;

public class ProductPriceDelegate {
    private SOAPConnection con = null;
    public ProductPriceDelegate(SOAPConnectionFactory cf) {
        try {
            // Create a connection
            // from the connection factory looked up.
            con = cf.createConnection();
        }
    }
}

```

① Business delegate

② Cache connection

③ ProductPriceDelegate constructor

```

    } catch(Exception e) {
        throw new IllegalStateException(
            "Unable to obtain JAXM connection.");
    }
}

/** Business method to obtain current pricing for a product.
 * @param productId The product's ID
 * @return Current price as a double
 */
public double getProductPrice(String productId)
    throws ProductPricingException {
    try {

        // Create a message factory from the connection
        MessageFactory mf = con.createMessageFactory();

        // Create a message from the message factory.
        SOAPMessage msg = mf.createMessage();

        // Message creation takes care of creating the SOAPPart
        SOAPPart sp = msg.getSOAPPart();

        // Retrieve the envelope from the soap part
        SOAPEnvelope envelope = sp.getSOAPEnvelope();

        // Create a soap header from the envelope.
        SOAPHeader hdr = envelope.createSOAPHeader();

        // Create a soap body from the envelope.
        SOAPBody bdy = envelope.createSOAPBody();

        SOAPHeaderElement
            quoteDate = hdr.createSOAPHeaderElement("QuoteDate",
                Namespace.getNamespace("q",
                    "http://www.mycompany.com/PriceQuote"));
        quoteDate.setMustUnderstand(true);
        quoteDate.addContent((new Date()).toString());

        // Add a soap body element to the soap body
        SOAPBodyElement gpp
            = bdy.createSOAPBodyElement("GetProductPrice",
                Namespace.getNamespace("m", "MyCompany-URI"));
        gpp.createSOAPElement("productId",
            Namespace.getNamespace("m", "MyCompany-URI"))
            .addContent(productId);

        // Set the soap header and soap body on the envelope.
        envelope.setSOAPHeader(hdr);
        envelope.setSOAPBody(bdy);

        // Create an endpoint for the recipient of the message.
        Endpoint endPoint = new Endpoint("www.mysupplier.com");

        // Get the price via RPC

```

Adds SOAP
header
element **4**

SOAP body
contains
both
methods
and
parameters

```
SOAPMessage priceMsg = con.call(msg, endPoint);

// Inspect the results to obtain price
String priceString
    = msg.getSOAPPart().getSOAPEnvelope()
        .getSOAPBody().getChild("GetProductPriceResponse",
            Namespace.getNamespace("m", "MyCompany-URI"))
        .getTextTrim();
return Double.parseDouble(priceString);
} catch(Exception e) {
    throw new ProductPricingException(productId,
        "Unable to obtain price.");
}
} // end getProductPrice()
}
```

- ❶ This class is a Business Delegate for using SOAP XML-RPC messaging to obtain product price information from our manufacturer.
- ❷ Once we obtain a connection to the messaging system, we cache it using the variable `con`. This allows us to avoid the overhead of obtaining the connection multiple times.
- ❸ When this class is instantiated, its first task is to connect to the messaging system. If that connection succeeds, then we cache the connection in the variable `con`. If it fails, then the Business Delegate is of no use to the client and it throws an exception.
- ❹ Though it is not necessary, we add a SOAP header element to our message to show how it is done. The element that we add to the message is today's date.

This code uses JAXM to create a synchronous message that obtains the price for a specific item. In section 4.3.2, we discussed a scenario in which an asynchronous SOAP message is sent from the manufacturer to the distributor because a product image was updated. In the next section, we examine the Java code necessary to receive and process that image.

Receiving asynchronous SOAP messages

Receiving asynchronous SOAP messages via JAXM must be done via the J2EE web container, since there is currently no JAXM support in the Message Driven EJB construct. To receive asynchronous SOAP messages, you need to implement a servlet to receive the inbound messages and route them to other components as appropriate. JAXM provides a base servlet called `JAXMServlet` for use in these situations. Listing 4.5 is a JAXM servlet to handle the asynchronous product image updates described in listing 4.3.

The manufacturer generates a SOAP message whenever a product image has been updated. When the message arrives at the distributor, a `JAXMServlet`'s `onMessage` method is called with a `JAXM Message` parameter. Our example servlet inspects the `Message` to obtain a reference to a `JDOM Element` that contains the method call parameters.

```
Element productIdElem =
    message.getSOAPPart()
        .getSOAPEnvelope()
            .getSOAPBody()
                .getChild("productId",
                    Namespace
                        .getNamespace("myco",
                            "http://schemas.mycompany.com/Product-Images"));
```

The `JDOM Element` is then inspected to determine the product identifier, type of image, and the identifier for the MIME block that contains the image data.

```
String productId
    = productIdElem.getAttributeValue("id");

String imageType
    = productIdElem.getAttributeValue("type");

String mimeTypeId
    = productIdElem.getChild("imageData")
        .getAttributeValue("href");
```

The attachments are then inspected to locate the image data and save it to a file. The `JAXM` API assumes that more than one attachment is present in any message, so we must iterate over a collection of one in this case.

```
Iterator attachmentIterator
    = message.getAttachments();

while (attachmentIterator.hasNext()) {
    AttachmentPart ap
        = (AttachmentPart) attachmentIterator.next();

    // match attachment's content id to the
    // id specified above
    if (mimeTypeId.equals(ap.getContentId())) {
        . . .
    }
}
```

Listing 4.5 contains the full source code for the `JAXM` servlet that accepts inbound SOAP messages.

Listing 4.5 A JAXM servlet for inbound SOAP messaging

```

import javax.xml.messaging.*;
import javax.xml.soap.*;
import javax.activation.DataHandler;
import org.jdom.*; ← JAXM is built on
import java.io.*;    top of JDOM
import java.util.*;

/**
 * A JAXM servlet that receives asynchronous
 * image updates for products from a supplier.
 */
public class ImageUpdateServlet extends JAXMServlet {
    public SOAPMessage onMessage(SOAPMessage message) {
        try {
            Element productIdElem =
                message.getSOAPPart()
                    .getSOAPEnvelope()
                    .getSOAPBody()
                    .getChild("productImage",
                        Namespace
                            .getNamespace("myco",
                                "http://schemas.mycompany.com/Product-Images"));

            // get the product ID corresponding to
            // the image file
            String productId
                = productIdElem.getAttributeValue("id");

            // get the image type
            String imageType
                = productIdElem.getAttributeValue("type");

            // get the attachment's MIME id
            String mimeTypeId
                = productIdElem.getChild("imageData")
                    .getAttributeValue("href");

            // go get the attached image data
            Iterator attachmentIterator
                = message.getAttachments();

            // iterate the attachments - should only be one
            while (attachmentIterator.hasNext()) {
                AttachmentPart ap
                    = (AttachmentPart) attachmentIterator.next();

                // match attachment's content id to the
                // id specified above
                if (mimeTypeId.equals(ap.getContentId())) {

```

Handles
image
update

Gets the SOAP block
containing the
product ID


```
        // obtain a data handler for the type of content
        // see javax.activation.DataHandler for details
        DataHandler dh = ap.getDataHandler();

        // store the binary image data someplace
    }
}

} catch(Exception e) {
    // log the error, panic, etc.
}

// asynchronous SOAP - no message to send back
return null;
}
}
```

- ① Handles the image update asynchronous message. This method returns null because no response is sent back to the message creator.

SOAP is an important development in the application integration space. Our discussion has been so detailed because we believe that it will play a significant role in your J2EE application in the future.

4.4 *Building web services in J2EE*

The idea of web services is taking the technology industry by storm, due in large part to vigorous support from software giants such as IBM, Sun Microsystems, and Microsoft. It is central to Microsoft's .NET strategy, as well as those of IBM, Sun, and numerous other companies. Since web services appear destined to succeed, we discuss building and using web services in the J2EE environment in this section.

As XML messaging technologies such as SOAP have gained momentum, moving distributed application architectures away from tightly coupled technologies such as CORBA and RMI has become possible. The web services architecture is a loosely coupled, service-oriented environment in which applications expose functionality to one another over the Web. This type of architecture maximizes the flexibility and interoperability of distributed applications. Because it is based on open standards such as XML and HTTP, it is completely vendor- and implementation-independent. A web service created using C++ and Microsoft's .NET development tools can be consumed by a Java component running in a J2EE container. And once these services become pervasive, new types of applications that aggregate a set of services into completely integrated, inter-enterprise distributed systems will be possible.

4.4.1 What is a web service?

A *web service* is some set of functionality made available to remote applications and services via the Internet. A web service is described in XML, using the Web Services Definition Language (WSDL). As shown later in this section, a WSDL description contains details about what the service is, where to find it on the Web, and how to interact with it. Once created, this description is registered in a well-known location. Figure 4.13 depicts a sample web service. There are currently two popular, competing standards in the web services registry space, the ebXML Registry and Repository and Universal Description, Discovery, and Integration (UDDI). Of these, UDDI is more general-purpose and is rapidly gaining the support of a majority of the industry.

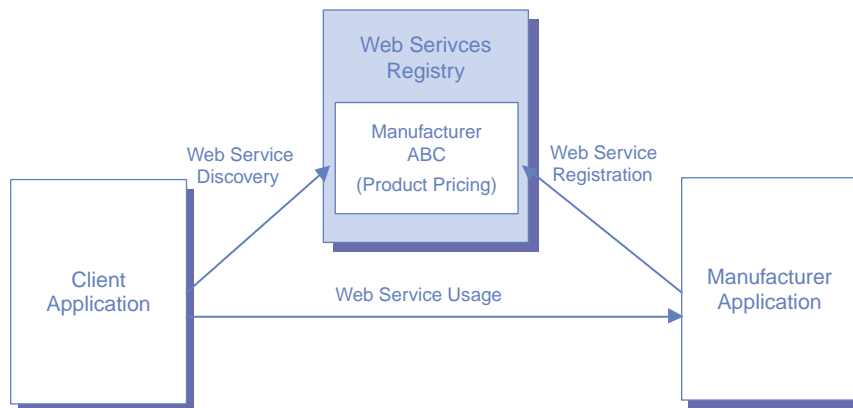


Figure 4.13 Basic structure of a web service

UDDI defines a system of interoperating service registries that collaborate over the Internet. Its operation is conceptually similar to the Domain Name System (DNS). In UDDI, businesses that provide web services register them with an official UDDI registrar. The registration is then propagated throughout the system, allowing any potential service consumer to search for and locate it via any UDDI engine.

Once a web service has been located, the interaction between producer and consumer takes place over standard Internet protocols such as HTTP, FTP, and SMTP. As described earlier, SOAP has an explicit binding to the HTTP protocol. This has made the use of SOAP for web service integration the de facto standard. Where secure interactions are required, HTTPS can be used instead.

Web service interfaces

A web service can be designed to use a one-way, message-based interface or a two-way, RPC-style interface. This is consistent with our JAXM examples from the previous section.

Message-based web services are said to be document-driven, meaning that only the data passed between the parties is important. Issues like timing and coordination of processing are irrelevant. Data-oriented Web services lend themselves to the message-based style. For example, the SOAP message from listing 4.3 is data-driven, and its web service would employ the message-based style.

RPC-style services are interface-driven, synchronous interactions. A web service that is process-oriented will use the RPC-style. For example, the SOAP interaction described in listing 4.1 is synchronous (process-driven), and would use the RPC-style.

4.4.2 Providing web services in J2EE

J2EE web services leverage the HTTP capabilities of servlets for sending and receiving messages. For RPC-style web services, you create a servlet that accepts the inbound request, interacts with the component providing the service, and returns an appropriate response. The most obvious choice for implementing a J2EE web service is a stateless session bean, since web services are self-contained RPC calls that do not maintain state across invocations.

The suggested architecture for a J2EE RPC-style web service is depicted in figure 4.14.

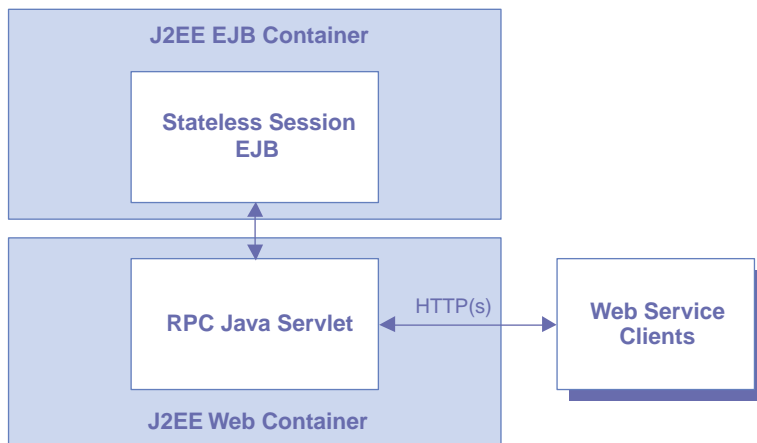


Figure 4.14 RPC-style web service in J2EE

Message-style web services

For message-style web services in J2EE, the inbound servlet can place messages on a JMS topic or queue to be processed asynchronously by a Message Driven EJB. If a response is required after the message has been processed, the Message Driven Bean can place the response on an outbound queue for delivery via an outbound servlet. This architecture is depicted in figure 4.15.

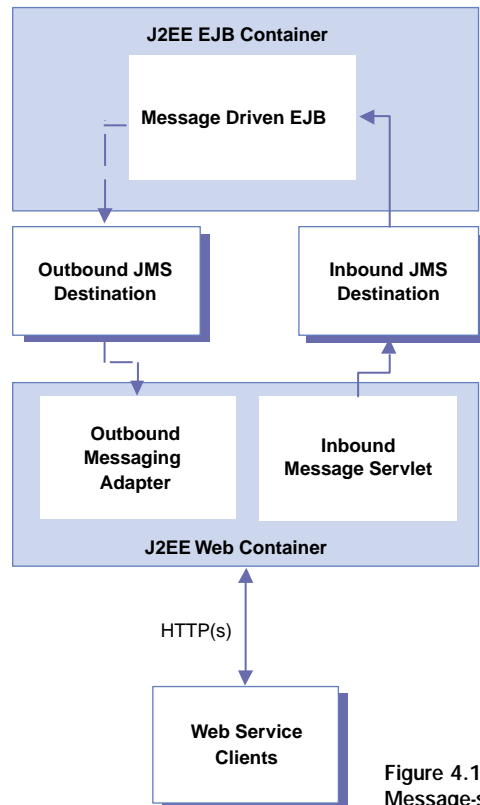


Figure 4.15
Message-style web service in J2EE

J2EE web services component model

Since J2EE web services are built upon open standards, vendor tools can generate much of the code and supporting XML descriptions of your web service automatically for you. This allows you to concentrate on the specific service being implemented, rather than the details of SOAP messaging and WSDL. For example, BEA's WebLogic product will generate the WSDL for your web

service and provide a client JAR file to be downloaded to remote users of the service. It additionally provides generic proxy servlets to receive and send web service messages using SOAP 1.1 with attachments. The only requirements placed upon you are that you implement your web services using the patterns discussed in the previous section (stateless session EJBs for RPC and Message Driven EJBs for message-style) and that you run those EJBs through a web service generation utility. This can make building web services much faster and easier. If you are using another J2EE server, consult your vendor's documentation for similar functionality.

As web services registries and the JAXR API mature, we expect to see automated tools for registering your web services in UDDI and similar systems as well. For now, the registration process is outside the scope of vendor implementations.

4.4.3 *Implementing our example web services*

To demonstrate the creation of J2EE web services, we implement both an RPC-style web service and a message-style web service. The environment that we have chosen for these examples to run in is BEA WebLogic version 6.1. Although there are many vendor products that provide support for web services, we have chosen WebLogic based in its popularity in the industry. An evaluation version of WebLogic can be downloaded at <http://www.bea.com>. Additionally, we use WebLogic in chapter 6 as an application environment for our case study. We do, however, make every attempt to keep our examples vendor- and product-independent. We note any area in which we use WebLogic-specific services.

An RPC-style web service

For the RPC-style example, we implement a web service that provides price updates to our distributor from our manufacturer. This is the web service that would have created the SOAP response message shown in listing 4.1. We require a stateless session bean that implements the `GetProductPrice` interface described in listing 4.1. This bean will contain a single business method that accepts a `String` representing a product identifier and returns a double representing the current price.

```
public double getProductPrice(String productId) {
```

The code for our session bean is shown in listing 4.6.

Listing 4.6 RPC web service session bean

```
import javax.ejb.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * A simple stateless SessionBean that implements
 * our product pricing, RPC-style web service.
 */
public class ProductPricingBean implements SessionBean {

    /**
     * Retrieve the current price of the specified product.
     *
     * @param productId The product to be priced
     * @return the price as a double
     * @exception EJBException if there is
     *             a communications or systems failure
     */
    public double getProductPrice(String productId) {
        return (new Double("9.95")).doubleValue();
    }

    // EJB specification stuff below
    private SessionContext ctx;

    public void ejbActivate() { }

    public void ejbRemove() { }

    public void ejbPassivate() { }

    public void ejbCreate () throws CreateException { }

    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }
}
```

Web Service method exposed to vendors

- 1 In the body of this method, the manufacturer's application would typically interact with other components, services, or data sources. For the purposes of this example, we assume that the product price is always \$9.95 USD.

This EJB can now be run through a vendor tool to produce the WSDL file and link it to a SOAP proxy servlet. But first we will create the message-style web service for product image updates. After developing the code for both services, we will create and examine WSDL files for both of them together.

SOAP data types

Note that, due to limitations of the SOAP encoding rules, only Java primitive types, their wrapper objects, and XML DOM objects can currently be used as parameter and return types to a web service. Table 4.2 shows the supported Java data types you can use in WebLogic web services as an example of this.

Table 4.2 Supported Java data types for WebLogic 6.1 web services

Java data type category	Examples
Primitive data types	int, short, long, double, float, Boolean
Wrapper classes for primitive types	java.lang.Integer, java.lang.Short, java.lang.Long, java.lang.Double, java.lang.Float, java.lang.Boolean
W3C classes representing XML documents, fragments, and elements	org.w3c.dom.Document, org.w3c.dom.Element
JavaBeans containing properties of the types listed in this table only.	A Bean defining two properties of type int and org.w3c.dom.Document
One-dimensional arrays of other data types listed in this table	A one-dimensional array of org.w3c.dom.Document objects

A message-style web service

For the message-based example, we implement a web service that provides asynchronous updates of images for a product catalog. This is the web service that would have produced the SOAP message shown in listing 4.3.

The implementation includes a Message Driven Bean that accepts the product image update message shown in listing 4.3. This EJB will be invoked whenever a message is placed on the JMS Destination we configure for the web service. Your J2EE vendor may provide a proxy servlet to handle the receipt of the message for you, similar to the RPC-style proxy servlet used in the previous example. However, the manner in which messages are placed on the queue can vary by vendor. For example, WebLogic 6.1 uses its own proprietary SOAP API to parse incoming messages and put their payloads on a JMS topic or queue. If you want to use the built-in proxy functionality for message-style web services, you may need to use the vendor's API to access your incoming messages.

Since we like our J2EE code to be vendor-free, our example implementation for the message-style web service uses JAXM to handle incoming messages. This requires extra work on our part to create and deploy the proxy servlet, but also gives us full control over the entire process. Listing 4.7 shows the Message Driven EJB that processes the product image updates. We do not

cover it step-by-step here, due to its similarity to the JAXM servlet in listing 4.5.

Listing 4.7 Message-style web service bean

```

import javax.ejb.*;
import javax.naming.*;
import java.rmi.RemoteException;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
import javax.jms.JMSEException;
import javax.xml.messaging.*; ← Uses JAXM to process
import javax.xml.soap.*;      message contents
import javax.activation.DataHandler;
// JAXM based on JDOM
import org.jdom.*;

import java.io.ByteArrayInputStream;
import java.util.Iterator;

public class ImageUpdateListenerBean
    implements MessageDrivenBean, MessageListener {

    private MessageDrivenContext ctx;
    private transient SOAPConnection jaxmCon; ← Connection to JAXM
                                              provider

    public void onMessage(javax.jms.Message msg) { ① Receives
                                                    message from
                                                    JMS queue

        TextMessage tm = (TextMessage) msg;
        try {
            String jaxmMsgString
                = (String) tm.getText();
            MessageFactory mf =
                jaxmCon.createMessageFactory();
            SOAPMessage message = mf.createMessage(
                (new ByteArrayInputStream(
                    jaxmMsgString.getBytes("UTF-8")))
                );
            // now inspect the SOAP message to
            // determine the product ID and obtain
            // the attached image file

            Element productIdElem =
                message.getSOAPPart()
                    .getSOAPEnvelope()
                    .getSOAPBody()
                    .getChild("productId",
                        Namespace
                            .getNamespace("myco",
                                "http://schemas.mycompany.com/Product-Images"));

```



```

String productId7
    = productIdElem.getAttributeValue("id");

String imageType
    = productIdElem.getAttributeValue("type");

String mimeTypeId
    = productIdElem.getChild("imageData")
        .getAttributeValue("href");
    Gets attachment's
    MIME id

Iterator attachmentIterator
    = message.getAttachments();
    Gets attached
    image data

while (attachmentIterator.hasNext()) {
    AttachmentPart ap
        = (AttachmentPart) attachmentIterator.next();
    if (mimeTypeId.equals(ap.getContentId())) {
        DataHandler dh = ap.getDataHandler();
        // store the binary image data someplace
    }
}
} catch (Exception ex) {
    // log, email, panic, etc.
}
}

// Other EJB and helper methods
// -----
public void ejbCreate () throws CreateException {
    try {
        getJAXMConnection();
    } catch (Exception e) {
        throw new CreateException(e.getMessage());
    }
}

private void getJAXMConnection()
throws Exception {
    if (jaxmCon == null) {
        Context namingCtx =
            new InitialContext();
        SOAPConnectionFactory myCF =
            (SOAPConnectionFactory) namingCtx.lookup(
                "java:comp/env/jaxm/JAXMConnFactory");
        jaxmCon = myCF.createConnection();
    }
}

private void releaseJAXMConnection() {
    if (jaxmCon != null) {
        try {
            jaxmCon.close();
        } catch (Exception e) { }
    }
}

```

```

    }

    public void ejbActivate() {
        try {
            getJAXMConnection();
        } catch (Exception e) {
            throw new IllegalStateException(e.getMessage());
        }
    }

    public void ejbRemove() { releaseJAXMConnection(); }

    public void ejbPassivate() { releaseJAXMConnection(); }

    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        this.ctx = ctx;
    }
}

```

- ❶ Receives messages from a JMS queue each time a product is updated. The `onMessage` method processes the JAXM message passed in from the queue.
- ❷ As we will see in listing 4.8, the JAXM `Message` is not directly serializable. Once we receive the message from the JMS queue, the first step is to rebuild the JAXM message.

The `JAXMServlet` that accepts the updates from our supplier and places them on the JMS Queue is shown in listing 4.8. When invoked, this servlet transfers the JAXM `Message` contents into a JMS `Message`. This is a bit complicated, because JAXM messages are not currently serializable.

```

ByteArrayOutputStream baos =
    new ByteArrayOutputStream();
message.writeTo(baos);
String jaxmMsgString = baos.toString("UTF-8");

```

After we have the message represented as a `ByteArrayOutputStream`, we can create a JMS `TextMessage` and place it on the queue, bound for our `Message Driven Bean`.

```

TextMessage msg =
    qsession.createTextMessage();
msg.setText(jaxmMsgString);
qsender.send(msg);

```

Listing 4.8 JAXM web service servlet

```
import javax.naming.*;
```

```

// explicit class imports to avoid
// name collisions between JAXM and JMS
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.QueueSender;
import javax.jms.Queue;
import javax.jms.TextMessage;

import javax.xml.messaging.*;
import javax.xml.soap.*;
// JAXM based on JDOM
import org.jdom.*;

import java.io.*;
import java.util.*;
import javax.servlet.*;

public class ImageUpdateServlet extends JAXMServlet {
    private QueueConnectionFactory qcFactory = null;
    private Queue queue = null;

    public void init(ServletConfig servletConfig)
        throws ServletException {
        try {
            Context naming = new InitialContext();
            qcFactory = (QueueConnectionFactory)
                naming.lookup(
                    "mycompany.jms.QueueConnectionFactory");
            queue = (Queue)
                naming.lookup(
                    "mycompany.jms.ProductImageUpdateQueue");
        } catch (Exception e) {
            throw new ServletException(e.getMessage());
        }
    }

    public SOAPMessage onMessage(SOAPMessage message) {
        try {
            // put the JAXM Message on the Queue
            ByteArrayOutputStream baos =
                new ByteArrayOutputStream();
            message.writeTo(baos);
            String jaxmMsgString = baos.toString("UTF-8");

```

JAXM
proxy
servlet
①

Creates naming
context

Handles
image
update
②

Dumps
object to
UTF-8 string
③

```

QueueConnection qcon =
    qcFactory.createQueueConnection();
QueueSession qsession =
    qcon.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);
QueueSender qsender =
    qsession.createSender(queue);
TextMessage msg =
    qsession.createTextMessage();

qcon.start();
msg.setText(jaxmMsgString); | Sends JAXM message
qsender.send(msg);
// clean up
qsender.close();
qsession.close();
qcon.close();

} catch(Exception e) {
    // log the error, panic, etc.
}
// asynchronous SOAP - no message to send back
return null;
}
}

```

Establishes queue
connection

- ❶ This class is a JAXM web services proxy servlet that receives asynchronous image updates for products from a supplier.
- ❷ The `onMessage` method handles the image update from the asynchronous SOAP message. The parameter that it receives is the JAXM Message. This method returns null because no response is sent back to the message creator.
- ❸ This code block dumps the object to a UTF-8 string. We do this before we put it on the JMS queue because JAXM Messages are not serializable.

Now that we have created both the RPC-style and message-Style web services, we need to publish them for clients. This involves generating WSDL that describes our new services.

Using WSDL to describe web services

For our new web services to be discovered and invoked by new business partners, we must first create a description of each using WSDL. This description provides details about what the web service does, how to invoke it, and where it can be found on the Internet. These WSDL descriptions are then published in one or more web service registries like UDDI.

Table 4.3 Information groups in WSDL

WSDL information type	Description
Data types	The <types> element allows you to derive new XML Schema data types for elements used in your WSDL descriptions.
Messages	The <message> element defines a one-way message, including its parameters, if any. An RPC-style web service with one operation would define two <message> elements, one for the request and one for the response.
Port types	Port types are abstract constructs that allow grouping of message definitions into operations. They define request/response interaction without regard to the specifics of the protocol used to exchange the messages (which is handled by <binding> information).
Bindings	A <binding> element maps a port type to a communication mechanism, like SOAP. These data add protocol-specific information to the port type information.
Web service instances	A <service> element describes an instance of a web service, including where it is located (URL), what its interface is (port type) and how to interact with it (binding).

A WSDL file contains five groups of information, as described in table 4.3. These data are layered on top of each other to provide a complete description of your web service. At the top layer, the <service> elements describe the web service(s) provided and their URLs. This information refers to and relies on the information from the <binding>, <portType>, and <message> elements in the document. The <types> construct allows you to define new XML Schema data types if needed.

Figure 4.16 visually depicts the relationship between elements in a WSDL file. The WSDL for our example RPC web service is shown in listing 4.9.

Listing 4.9 WSDL for the product pricing web services

```
<?xml version="1.0"?>
<definitions
  targetNamespace="java:examples.chapter4.webservices.rpc"
  xmlns:tns="java:examples.chapter4.webservices.rpc"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
>
  <types>
    <schema
```

```

        targetNamespace='java:examples.chapter4.webservices.rpc'
        xmlns='http://www.w3.org/1999/XMLSchema' >
        <!-- Derived XML data types would go here -->
    </schema>
</types>

<!--
    Inbound/Outbound message definitions for our RPC service
-->
<message name="getProductPriceRequest">
    <part name="productId" type="xsd:string" />
</message>
<message name="getProductPriceResponse">
    <part name="return" type="xsd:double" />
</message>

<!--
    The definition of the web service port type.
    This binds the messages above together into
    an RPC request/response "operation."
-->
<portType name="ProductPricingPortType">
    <operation name="getProductPrice">
        <input message="tns:getProductPriceRequest"/>
        <output message="tns:getProductPriceResponse"/>
    </operation>
</portType>

<!--
    A binding of our new port type to SOAP/HTTP.
-->
<binding
    name="ProductPricingBinding"
    type="tns:ProductPricingPortType">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getProductPrice">
        <soap:operation soapAction="urn:getProductPrice"/>
        <input>
            <soap:body use="encoded" namespace='urn:ProductPricing'
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
        </input>
        <output>
            <soap:body use="encoded" namespace='urn:ProductPricing'
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
        </output>
    </operation>
</binding>

<!--
    The actual web service definition, built from the
    message, port, and binding information above.
-->

```

```

    This is where the actual URL for the service is
    specified.
-->
<service name="ProductPricing">
  <documentation>
    This service accepts a product ID from our catalog and
    returns its current list price.
  </documentation>
  <port name="ProductPricingPort" binding="tns:ProductPricingBinding">
    <soap:address
location="http://localhost:7001/productPricing/productPricingURI"/>
  </port>
</service>
</definitions>

```

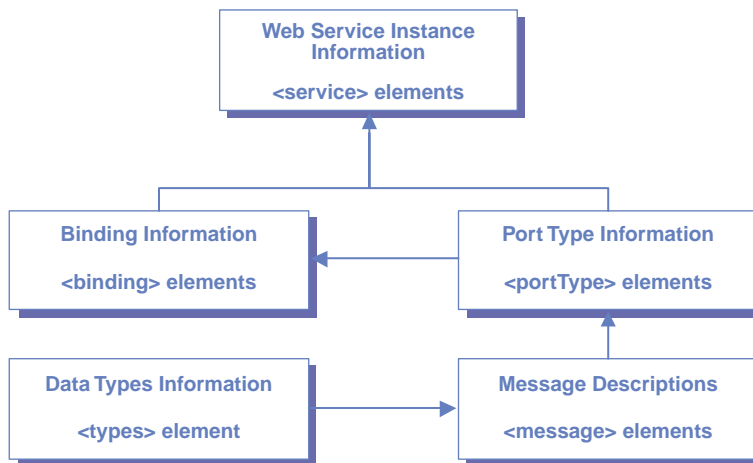


Figure 4.16 Data relationships in WSDL

Registering web services in UDDI

Now that the web service implementations exist and the WSDL has been generated, all that is left to do from the web service provider side is to register the services with a UDDI registry. Both commercial and open source development tools are becoming available to automate this task. If our web services were to be publicly available, we would need to register with an official UDDI registrar. Otherwise, we could purchase a UDDI registry product and run UDDI on an internal network. See <http://www.uddi.org> for a current list of UDDI registrars and UDDI server providers.

Once we have decided where to register the web services, the task of connecting to the registry and registering them can be automated using an API. In the future, the JAXR API should be available to make this automation vendor-independent. For now, you must implement such functionality using a third-party API such as the Free Software Foundation's pUDDIng library. Such APIs provide the ability to connect to a UDDI registry server and register (as well as discover) your web services. You might consider writing a parameter-driven utility application using pUDDIng or another UDDI tool to automate registrations, especially if you plan to create numerous web services in the future. Information about pUDDIng and other tools can be found through the UDDI URL listed in the preceding paragraph.

4.4.4 *Consuming web services in J2EE*

Accessing web services from J2EE is similar to other types of systems integration tasks with one extra step. That step has to do with locating the web service in a registry before connecting to it. This can be done directly via APIs such as pUDDIng or (in the future) JAXR. It is also likely that J2EE connectors will soon be built to make this service discovery process transparent to the application developer. You may, for instance, obtain a web service connection factory via JNDI in your code and configure it externally, similar to JMS Destinations, JDBC data sources, and other resources in J2EE.

Many of the design patterns discussed in chapter 2 can be useful in terms of web service interaction from an EJB container. You might use the Business Delegate pattern to encapsulate the complexities of making a SOAP RPC call. This delegate might use a combination of JAXM and JAXR to locate a web service, build a SOAP message, and make an RPC call to a web service. Such an implementation would be similar to the Business Delegate we created in listing 4.4. In cases where you need to model a set of web service operations as an internal application entity, you might also consider using the Aggregate Entity or Data Access Object patterns with web service-aware helper classes.

4.4.5 *J2EE web services and Microsoft .NET*

All the major players in the software development industry have initiatives surrounding web services. Sun Microsystems, IBM, and Microsoft are each touting the web services architecture as a key part of their business development strategies for years to come. This begs the question, what issues can I expect to encounter when trying to integrate my J2EE system with a non-Java web service implementation? More specifically, will I really be able to actually

integrate with a Microsoft-based environment just because we all use SOAP and HTTP to communicate?

The short answer is yes, with some work potentially required. It is true that SOAP and HTTP bring us a lot closer to an open services environment. However, there are some details to be tripped over when integrating a foreign web service into your environment. These details concern the encoding of the actual data carried within SOAP messages, not the messages themselves.

Mapping primitive data types between a Java and COM environment is not particularly challenging, but the same cannot be said about such complex types as software objects. To communicate, both parties in a SOAP communication must speak the same encoding language, having agreed to the structure and interpretation of the data payload in each message. This can be more complicated than it appears. For example, the WebLogic 6.1 SOAP implementation can only handle the types shown in table 4.1. This means that you must somehow serialize any complex object to XML or a JavaBean containing its constituent primitive types as properties for transport.

In simple scenarios (which most web service interactions should be), communicating with a .NET web service should be easy. This simple scenario refers to the passing of data types explicitly handled by the SOAP encoding rules. For data types that fall outside these bounds, an agreed-upon encoding mechanism and the software objects to do the encoding and decoding between platforms will be required.

4.5 *Summary*

This chapter began by describing systems integration, the process of gluing individual applications and services together to create robust distributed systems. We discussed the four general architectural patterns for integrating independent systems. These patterns describe the four ways in which two systems communicate: through data, messages, remote procedures, and object interfaces. After discussing general integration patterns, we examined a proven approach to systems integration called constraint-based modeling.

After understanding the systems integration process, the remainder of the chapter focused on the ways in which XML technologies can make integration easier and more robust. We outlined the current XML standards and technologies that can be implemented in each of the four architectural patterns, including some of the key Java/J2EE APIs that implement them.

Sections 4.2 and 4.3 focused on SOAP and web services. These sections included brief introductions to the technologies in general, along with

examples of their implementation in Java. Much of your XML-based systems integration work in the future will likely involve web services, so investing the time to master SOAP, UDDI, and WSDL should be well worth it.

For another look at how web services are constructed and used in J2EE, you need only look to the case study in chapter 6. In that chapter, we combine our understanding of systems integration with the concepts from chapters 3 and 5 to create an end-to-end example of a J2EE/XML application that is integrated into its environment via web services.



User interface development

This chapter

- Demonstrates use of XSLT with Java servlets
- Compares pure J2EE user interface development with an XML approach
- Shows how to build multidevice and multilocale user interfaces
- Covers XML web publishing frameworks

Creating a robust presentation layer for your J2EE application is a challenging endeavor. This is so because the vast majority of J2EE applications are web-based, also known as “thin-client”, applications. In this chapter, we examine some emerging challenges in J2EE user interface design and discuss ways you might use XML technology to overcome them.

We begin by exploring the nature of thin-client user interface development and the challenges you are likely to face when building and maintaining an advanced presentation layer. We then examine the “pure” J2EE approach to building such a layer to see where the current J2EE architecture is lacking.

The remainder of the chapter focuses on overcoming the limitations of the pure J2EE approach using XSLT technology. First we develop an XSLT-based presentation layer from scratch. Then we use XSLT from within a web publishing framework to discover the benefits and drawbacks of using a third party API.

The goal of this chapter is not to convince you that one architecture or product is superior to another. We wish only to make you aware of the available options, see them in action, and understand the positive and negative aspects of taking each approach.

5.1 *Creating a thin-client user interface*

In this chapter, we focus almost exclusively on web-based, thin-client distributed applications. Before diving into the details of overcoming challenges associated with these types of applications, we should take a moment to discuss what they are and why building interfaces for them is so difficult.

DEFINITION A *thin-client application* is one in which the server side is responsible for generating the user interface views into the application. The client side usually consists only of generic rendering software, e.g., a web browser.

If you have built web-based applications before, you are no doubt familiar with the thin-client architecture. Until recently, the burden of generating the user interface for your application on the server side was not too difficult. Two relatively recent developments, however, are now making the development and maintenance of your presentation layer components more challenging.

5.1.1 *Serving different types of devices*

The first new challenge relates to the ongoing proliferation of web-enabled devices. It seems as though anything that runs on electricity these days now has a web browser on it. These “smart” devices include cell phones, PDAs, and refrigerators. (That only sounds like a joke.) The trouble with these smart devices is that they are usually quite dumb, virtually light years behind the current version of your favorite computer-based Internet browser. Some understand a subset of the HTML specification, and others require an entirely separate markup language. An example of the latter is the WAP-enabled cell phone, which requires web pages to be rendered using the Wireless Markup Language (WML).

DEFINITION WML is an XML-based presentation markup language specifically designed for web browsing on cellular telephones. These phones typically feature small display areas and very limited rendering capabilities. WML arranges content into a *deck of cards* (pages), each of which should contain very little data and uncomplicated navigation.

Creating a single, integrated user interface that can render content for various types of devices is difficult to do using the traditional J2EE (or ASP, or Cold Fusion, etc.) approach. User interface components in these technologies were not designed to change the entire structure of the generated content based on the type client requesting it. This means that, in order to serve web browser *and* cell phone client types from your J2EE web application, you must develop and maintain two separate presentation layers. One layer would generate an HTML interface, and the other a WML interface. Some updates to the application would require updates to *both* user interfaces. If you think managing two interfaces is scary, consider supporting three more types of client devices in the same application.

5.1.2 *Serving multiple locales*

The other unpleasant possibility concerns serving content to users in different locales. The obvious challenge here is to determine the end user’s preferred language and render the presentation in that language. This means having some mechanism to translate your application’s user interface between natural languages, either on the fly or offline. Offline translation can be expensive and time consuming, but its result is far less likely to be unintelligible. Real-time translation has been done poorly many times, but never done well as far as we know. Also consider that everything might need to be translated, including

text embedded in images and static text that might otherwise be embedded directly in your JSP.

The second, perhaps less obvious, dimension to the multiple locale problem has to do with cultural differences. Beyond the textual content of the page, it may be appropriate and advantageous to change the entire structure of your user interface for a new locale. You might want to change color schemes, page layouts, or even the entire navigational flow of the interface. The reasons for these types of changes have to do with psychology and marketing and other ungeeklike topics that we need not address here. In this chapter, we concentrate on how to satisfy these types of requirements, assuming away the reasons.

If you have tried to serve multiple languages and/or locales effectively using a pure J2EE presentation layer, you already appreciate the magnitude of the challenge. If not, we highlight its scariest features in the remainder of this chapter. What inevitably results from attempting to meet this requirement in the traditional approach is a large collection of mostly redundant, often misbehaved JSPs and servlets that are a nightmare to maintain and extend.

5.1.3 *An example to work through*

To really get at the heart of the matter and see how XML can help overcome the thin-client blues, let us look at some example requirements. For the sake of ubiquity and clarity, we examine a relatively simple user interface and a single point of functionality in a fictitious application. The application is a stock-trading web site, and the functional point to be explored is the rendering of a specific user's "watch list" of stock prices. Constraining our example to one small functional point allows us to concentrate on dynamically changing the user interface without bogging down in other details.

Here are the requirements that concern us in this chapter:

- The watch list page must be viewable via PC-based web browsers and WAP-enabled cell phones.
- The watch list page is rendered for two locales, the United States and Great Britain. Each locale requires its own specific interface with appropriate textual messages.
- The price for each stock listed on the watch list should be expressed in the user's local currency, USD (United States dollars) for the U.S. and GBP (British pounds) for the U.K.

You can see from the requirements above that our watch list user interface must consist of four pages. Figure 5.1 is an HTML page rendered for users in the United States.



Figure 5.1
Stock watch list
HTML page—
United States

Figure 5.2 is an HTML page rendered for users in Great Britain.



Figure 5.2 Stock watch list HTML page—Great Britain

To buy this book

Figure 5.3 is a WML page rendered for users in the United States.



Figure 5.3
Stock watch list WML page—United States

Figure 5.4 is a WML page rendered for users in Great Britain.



Figure 5.4
Stock watch list WML page—Great Britain

We chose the United States and Great Britain as the locales for this example to avoid the requirement of a second language on your part. In the remainder of this chapter, we create and recreate these four pages using a variety of techniques. In section 5.2, we use only the capabilities of J2EE and see where they fail us. In sections 5.3 and 5.4, we bring XML technology to the rescue and create a unified interface for this page that supports multiple device types and locales.

5.2 *The pure J2EE approach*

Before we discuss any new, XML-based architectural models for user interface creation, it is important to understand why we might need them. After all, we

do not want to fix something that is not broken nor overcomplicate an application unnecessarily. In this section we first explore what the pure J2EE approach has to offer and see some of its limitations when using it to develop our example pages.

5.2.1 The J2EE presentation tool kit

The various J2EE presentation layer components are summarized in table 5.1. All of these components, with the exception of applets, execute in the J2EE web container on the server side of a client/server application. These applications are almost always thin-client, web-based ones. In such an architecture the J2EE server-side components are designed to collaborate with one another in an adaptation of the Model-View-Controller (MVC) software design pattern.

Table 5.1 J2EE presentation framework components

J2EE component	Purpose
Applets	Java components downloaded and run in client browsers.
Servlets	Java components invoked through a client request to perform server-side processing and generate a response.
Java Server Pages (JSPs)	Response page templates that allow in-line inclusion of Java code with static content.
JavaBeans	Component framework for creating reuse in your applications.
JSP custom tags	Classes that enable complex logic to be referenced by a simple tag within a JSP. Often used to iterate over processing result sets and dynamically generate UI structures like HTML tables.
Filters	Server side components that intercept requests for pre and post processing.

The MVC presentation architecture is intended to decouple application data (*model*) from its rendering (*view*) to produce a more modular system. The *controller* is a traffic cop of sorts, managing the relationship between the views and the model. The two main goals of the MVC are:

- To enable multiple views of the same information
- To reuse the components used to generate the interface

When MVC is implemented correctly, individual components are independent of one another. A clock is a perfect example of this. The model, or mechanism for keeping time, functions the same regardless of the use of a digital or analog view.

In a J2EE presentation layer, one or more servlets usually act as the controller. They accept requests from clients, perform some processing on the model, and return an appropriate view. In some simple and other poorly designed systems, the servlet itself may render the view. This involves embedding static content like HTML tags directly in the servlet code, and is frowned upon by all of us for its shortsightedness.

A far superior approach is to have your controller servlet select an appropriate JSP as the view and forward processing to it. JSPs are far more maintainable and can make use of custom tags and JavaBeans to decouple much of the logic from the static content in your pages. This is the preferred presentation layer architecture referenced in the JSP specification (where it is called Model 2). Figure 5.5 illustrates the overall concept.

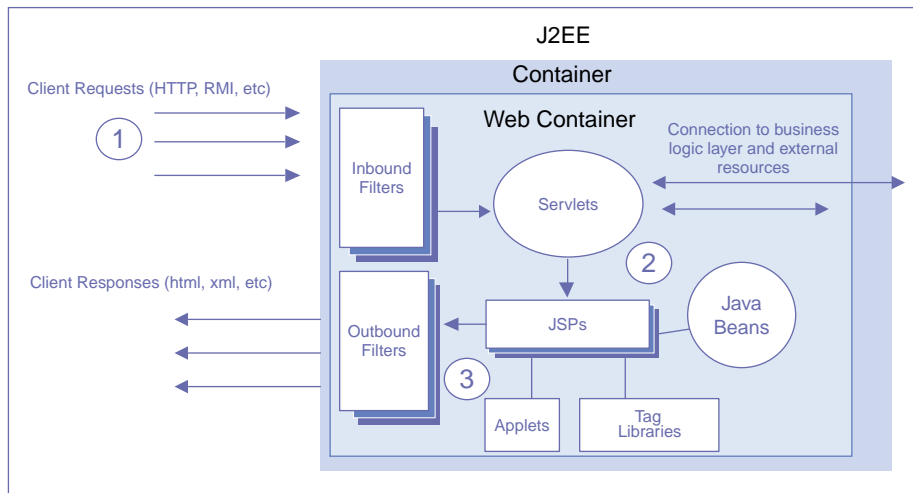


Figure 5.5 The J2EE MVC architecture

5.2.2 *Issues in J2EE MVC architecture*

The J2EE presentation framework is an adaptation of MVC for thin-client applications. In most cases, however, it does not succeed in fully decoupling presentation logic from data in your user interface.

DEFINITION *Presentation logic* refers to the programming elements that control the dynamic generation of the user interface. This term includes the programming variables, statements, and control structures used in your JSP pages.

In this section, we explore some of the goals and challenges common to virtually all pure J2EE presentation layer implementations, one by one.

Enforcing the separation of logic and content

Servlets and JSPs can be implemented in many different ways. In figure 5.5, we saw the Model 2 architecture in which JavaBeans are used as the model, servlets as the controller, and JSPs as the view.

Consider the following request-response flow:

- 1 A web request is intercepted by an inbound filter for preprocessing, authentication, or logging and then redirected to a servlet.
- 2 The servlet performs lightweight processing and interacts with your application's business logic layer, finally forwarding execution to a JSP for rendering.
- 3 The JSP uses a combination of JavaBeans and custom tags to render the appropriate presentation and then forwards the response to an output filter for any postprocessing before the response is returned to the user.

The success of any MVC implementation relies on the clear separation of roles (model, view, and controller) among components. When strictly enforced, the reusability and maintainability of each individual component is greatly enhanced. When not enforced, the attempted MVC architecture makes an already complicated system even harder to maintain and extend.

Limitations of template languages

JSPs are dynamic page templates that contain a combination of logic and data. They were modeled in a similar fashion to the other dynamic template languages, such as Active Server Pages and Cold Fusion, as an enabler to develop dynamic HTML pages. Completely separating code and data in such template languages is difficult, and at times impossible. For example, even the following three-line JSP contains page structure, static content, and code all mixed together.

```
<html>
<h1>Hello <%=request.getParameter(uName)%> </h1>
</html>
```

When JSP was originally developed, it was heavily criticized for embedding Java code directly into HTML pages (as done above). When data and code are collocated, there is inevitably contention between the HTML author and the JSP developer trying to work on the same source file. For enterprise applications,

there is usually an obvious need to separate the roles of UI designer and code hound. To solve this problem, the JSP specification evolved to include the use of custom tag libraries and JavaBeans. These additions freed JSPs of much Java code by encapsulating complex presentation logic within tags and data structures within JavaBeans. XML tags referring to these other components now replace much of the old JSP code.

While the amount of code present in JSPs decreased, the complexity of the overall architecture increased. In particular, tags often contain a combination of presentation markup tags and Java code, making some pages more difficult to maintain. The page designer cannot be expected to modify and recompile tags to accomplish a simple HTML change; and finding the code that needs to change can be tedious. A secondary issue with tags and JavaBeans in JSP is one of enforcement. Developers often opt for the path of least resistance, continuing to embed Java code directly in JSPs.

Code redundancy

As we discussed in section 5.1, the advent of multidevice and internationalization requirements for J2EE applications puts much more strain on the pure J2EE architecture. As we demonstrate in the next section, building our stock quote example pages requires between two and four individual JSP pages. As we increase either the number of locales or the number of client device types being serviced, the number of JSP pages proliferates. Consider the burden of developing and maintaining a set of 100 JSP pages to satisfy five functional requirements. Think about making the same change in 20 different JSP pages, testing them individually, and then regression testing the entire application. If the change is to logic, you hopefully have encapsulated it in a tag or a JavaBean. If the change is to the presentation layout or flow, you are in a world of hurt.

5.2.3 *Building our example in J2EE*

Enough discussion. It is time to see the issues in action. To service requests for our example watch list page using only J2EE, we require the following components:

- A servlet to accept the client request and interact with our application logic layer to obtain a list of stock quotes.
- A JSP for each device type for which we need to render the list.

Handling the request

We require a servlet to accept watch list user requests, obtain the watch list, and select a view with which to render the list. In this J2EE-only scenario, our servlet will forward the watch list to one of four JSPs, depending on the client device type the user is connecting from and the user's locale.

The first step our servlet will take is to obtain the user's stock quote list. We assume that our application has stored the user's unique identifier as a `String` in the `HttpSession` object. After retrieving that identifier, we make a call to the application logic layer to obtain the stock quote list as a JDOM value object.

```
HttpSession session = request.getSession(false);
ListBuilder builderInterface = ...
// ... validate session object exists

String userId = (String) session.getAttribute("userId");
org.jdom.Document quoteList = builderInterface.getWatchList(userId);
```

We then wrap the `quoteList` in a JavaBean object and store it in the request, for later retrieval by the JSP component. We go over the bean code later in this section.

```
XMLHelperBean helper = new XMLHelperBean(quoteList);
session.setAttribute(helper, helper);
```

The final step is the most involved. We must determine which of our four JSPs will render the response for the user. To do this, we must determine the user's device type and locale preference. First we develop a method to determine the device type, using the `User-Agent` HTTP header.

```
private String getOutputType(HttpServletRequest request) {
    String userAgent = request.getHeader("User-Agent");
    // compare to list of WAP User-Agents
    // for simplicity, we'll only try one here
    if (userAgent.indexOf("UP.Browser") >= 0)
        return "wml";
    return "html";
}
```

In a real-world scenario, this servlet would maintain a dictionary of known user-agents and their associated MIME content types. For the example, we only output WML if someone is using the `UP.Browser` phone browser. Calling the above method will set the output format. Now we require a method to choose a JSP based on the output format and the user's locale. For that, we use the `Accept-Language` HTTP header(s), which contains an ordered list of preferred locales from the user's browser settings.

```

private String getForwardURL(
    HttpServletRequest request, String outputType) {
    String result = null;
    if (outputType.equals("html"))
        result = "/watchlist/watchlist.html.en_US.jsp";
    else
        result = "/watchlist/watchlist.wml.en_US.jsp";
    Enumeration locales = request.getHeaders("Accept-Language");
    while (locales.hasMoreElements()) {
        String locale = (String) locales.nextElement();
        if (locale.equalsIgnoreCase("en_GB"))
            if (outputType.equals("html"))
                return "/watchlist/watchlist.html.en_GB.jsp";
            else
                return "/watchlist/watchlist.wml.en_GB.jsp";
    }
    return result;
}

```

The foregoing method will choose the appropriate JSP from the four we will develop shortly. Now all that is left to do is make use of these methods in our servlet's request handling method.

```

String outputType = getOutputType(request);
String forwardURL = getForwardURL(request, outputType);
context.getRequestDispatcher(forwardURL).forward(request, response);

```

The complete code for our new servlet is shown in listing 5.1.

Listing 5.1 The watch list JSP servlet

```

import org.jdom.*;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * The stock watchlist servlet with JSP
 */
public class WatchListJSPServlet extends HttpServlet {

    private ListBuilder builderInterface = new ListBuilder();
    private ServletConfig config;
    private ServletContext context;

    public WatchListJSPServlet() { super(); }

    public void init(ServletConfig config)
        throws ServletException {
        this.config = config;
        this.context = config.getServletContext();
    }
}

```

```

}

public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
    // get userid from HttpSession
    HttpSession session = request.getSession(false);
    if (session == null) {
        context.getRequestDispatcher("/login.jsp")
            .forward(request, response);
        return;
    }

    String userId = (String) session.getAttribute("userId");
    Document quoteList = builderInterface.getWatchList(userId);

    XMLHelperBean helper =
        new XMLHelperBean(quoteList);
    request.setAttribute("helper", helper);

    String outputType = getOutputType(request);
    String forwardURL =
        getForwardURL(request, outputType);

    context.getRequestDispatcher(forwardURL)
        .forward(request, response);
}

private String getOutputType(HttpServletRequest request) {
    String userAgent = request.getHeader("User-Agent");
    // compare to list of WAP User-Agents
    // for simplicity, we'll only compare one here
    if (userAgent.indexOf("UP.Browser") >= 0)
        return "wml";
    return "html";
}

private String getForwardURL(HttpServletRequest request,
                             String outputType) {
    String result = null;
    if (outputType.equals("html"))
        result = "/watchlist/watchlist.html.en_US.jsp";
    else
        result = "/watchlist/watchlist.wml.en_US.jsp";
    Enumeration locales = request.getHeaders("Accept-Language");
    while (locales.hasMoreElements()) {
        String locale = (String) locales.nextElement();
        if (locale.equalsIgnoreCase("en_GB"))
            if (outputType.equals("html"))
                return "/watchlist/watchlist.html.en_GB.jsp";
            else
                return "/watchlist/watchlist.wml.en_GB.jsp";
    }
}

```

User must be
logged in and have
an HttpSession

Store the document in a
JavaBean for later retrieval

Find the right JSP to
render the view

Forward the request and
response for rendering


```
        return result;
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Obtaining and using XML data

Our watch list servlet makes use of a `ListBuilder` component, the code for which you can find on the web site for this book (<http://www.manning.com/gabrick>). The `ListBuilder` returns a JDOM document containing a list of stock quotes in XML format. Listing 5.2 shows the XML data set we are using in the example. It contains three price quotes for stocks on my watch list. Note that multiple `price` nodes are contained within each `quote` element, each for a different currency. This will enable us to display prices in the user's native currency when the page is generated.

Listing 5.2 Stock quote XML data set

```
<?xml version="1.0"?>
<quote-list date="Nov. 4, 2001" time="9:32 AM EST">
  <customer first-name="Kurt" last-name="Gabrick" id="9999"/>
  <quote symbol="SRMC" name="Sierra Monitor Corporation">
    <price amount="2.00" currency="USD"/>
    <price amount="1.05" currency="GBP"/>
    <price amount="4000.00" currency="MXP"/>
  </quote>
  <quote symbol="IBM" name="International Business Machines">
    <price amount="135.00" currency="USD"/>
    <price amount="67.75" currency="GBP"/>
    <price amount="230000.00" currency="MXP"/>
  </quote>
  <quote symbol="ORCL" name="Oracle Corporation">
    <price amount="15.00" currency="USD"/>
    <price amount="7.75" currency="GBP"/>
    <price amount="30000.00" currency="MXP"/>
  </quote>
</quote-list>
```

After retrieving the JDOM document, we wrap it in a JavaBean component. The reason for this is to keep the XML manipulation code out of our JSP. By

storing this JavaBean in the `HttpRequest` object, we make it available to whichever JSP renders the view. The code for this bean is given in listing 5.3. Notice that the bean is also a custom tag, extending `javax.servlet.jsp.tagext.TagSupport` and implementing the `doStartBody` method. This allows us to dynamically generate the watch list presentation markup within the bean and eliminate all code from our JSPs.

Listing 5.3 The XMLHelper JavaBean/tag

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.util.*;

import org.jdom.*;

public class XMLHelperBean extends TagSupport {

    private String firstName;
    private String lastName;
    private String quoteTime;
    private String quoteDate;
    private Vector quotes = new Vector();

    private boolean useLinks=false;
    private String currency = "USD";

    public XMLHelperBean(Document doc) {
        Element root = doc.getRootElement();
        this.quoteTime = root.getAttributeValue("time");
        this.quoteDate = root.getAttributeValue("date");
        Element customer = root.getChild("customer");
        this.firstName = customer.getAttributeValue("first-name");
        this.lastName = customer.getAttributeValue("last-name");
        // build quote list
        List quoteElements = root.getChildren("quote");
        Iterator it = quoteElements.iterator();
        while (it.hasNext()) {
            Element e = (Element) it.next();
            Quote quote = new Quote();
            quote.symbol = e.getAttributeValue("symbol");
            quote.name = e.getAttributeValue("name");
            List priceElements = e.getChildren("price");
            Iterator it2 = priceElements.iterator();
            while (it2.hasNext()) {
                Element pe = (Element) it2.next();
                quote.prices.put(
                    pe.getAttributeValue("currency"),
```

JavaBeans
properties

Custom tag
attributes

```

        pe.getAttributeValue("amount")
    );
    }
}

public String getFirstName() {
    return firstName;
}

public String getLastName() {
    return lastName;
}

public String getQuoteTime() {
    return quoteTime;
}

public String getQuoteDate() {
    return quoteDate;
}

// supplied only for consistency with
// JavaBeans component contract - inoperative
public void setFirstName(String s) {}
public void setLastName(String s) {}
public void setQuoteTime(String s) {}
public void setQuoteDate(String s) {}

public void setUseLinks(String yesno) {
    if (yesno.equalsIgnoreCase("yes"))
        useLinks = true;
    else
        useLinks = false;
}

public boolean getUseLinks() { return useLinks; }

public void setCurrency(String currency) {
    this.currency = currency;
}

public String getCurrency() { return currency; }

public int doStartTag() {
    try {
        JspWriter out = pageContext.getOut();
        out.print("<tr><th>Stock Symbol</th>");
        out.print("<th>Company Name</th><th>Last Price</th>");
        if (useLinks)
            out.print("<th>Easy Actions</th>");
        out.print("</tr>");
        for (int i = 0; i < quotes.size(); i++) {
            Quote q = (Quote) quotes.get(i);
            out.print("<tr><td>");

```

← Dynamically builds
the watch list table in
either HTML or WML

Instead, we decided to write four very specific and simple JSPs, one for each permutation of output formats and locales we need to service. See listings 5.4 through 5.7. Each JSP will obtain a handle to the `XMLHelper` JavaBean that we stored in the request object to render its output.

Listing 5.4 The U.S. English, HTML watch list JSP

```

<jsp:useBean name="helper" scope="page"
             class="examples.chapter5.XMLHelperBean" />
<%@ taglib uri="example.tld" prefix="helperTag" %>
<html>
<head><title>Your Watch List</title></head>
<body>

<h1>Your Stock Price Watch List</h1>
<h3>
  Hello,
  <jsp:getProperty name="helper" property="firstName" />!
</h3>

<h3>
  Here are the latest price quotes for
  your watch list stocks.
</h3>

<p><i>
  Price quotes were obtained at
  <jsp:getProperty name="helper" property="quoteTime" />
  on
  <jsp:getProperty name="helper" property="quoteDate" />.
</i></p>

<table cellpadding="5" cellspacing="0" border="1">
<helperTag:printData useLinks="yes" currency="USD" />
</table>
</body>
</html>

```

Recovers the JavaBean from the request object

Specifies prefix for use as a tag

Invokes the `doStartTag()` of OUR `XMLHelperBean`

- ❶ Because we specify `useLinks=yes`, our custom tag prints an HTML table consisting of four columns with links to buy and sell individual stocks. Setting `currency=USD` will print all stock prices in U.S. dollars.

The JSP that produces the U.S. English, HTML version of our page is shown in listing 5.4, with its British counterpart given in listing 5.5. Note both the

similarities and the differences between the two. The U.S. page uses a bit less formal language and expresses prices in USD. The British version is more formal and shows pricing in British pounds. However, they both use the identical data source and neither has any Java code in it.

Listing 5.5 The British English, HTML watch list JSP

```
<jsp:useBean name="helper" scope="page"
             class="examples.chapter5.XMLHelperBean"/>

<%@ taglib uri="example.tld" prefix="helperTag" %>

<html>
<head><title>Your Watch List</title></head>
<body>

<h1>Your Stock Price Watch List</h1>

<h3>
  Greetings, Mr.
  <jsp:getProperty name="helper" property="lastName"/>!
</h3>

<h3>
  Here are the latest prices on
  your stocks of interest.
</h3>

<p><i>
  Price quotes as of
  <jsp:getProperty name="helper" property="quoteTime"/>
  on
  <jsp:getProperty name="helper" property="quoteDate"/>.
</i></p>

<table cellpadding="5" cellspacing="0" border="1">
  <helperTag:printData useLinks="yes" currency="GBP"/>
</table>
</body>
</html>
```

Now all that remains is to develop the WML versions of the watch list page. These pages contain a bit less textual information and do not have links to directly buy individual stocks. If you need to refresh your memory on what each of these pages looks like, flip back to figures 5.1 through 5.4.

Listing 5.6 The U.S. English WML watch list JSP

```

<jsp:useBean name="helper" scope="page"
    class="examples.chapter5.XMLHelperBean" />
<%@ taglib uri="example.tld" prefix="helperTag" %>

<wml>
  <card id="main" title="Your Watch List">
    <do type="accept" name="do-back" label="Back">
      <go href="http://www.exampleco.com/home.wml" />
    </do>
    <do type="accept" name="do-buy" label="Buy Shares">
      <go href="http://www.exampleco.com/buyShares" />
    </do>
    <do type="accept" name="do-sell" label="Sell Shares">
      <go href="http://www.exampleco.com/sellShares" />
    </do>
    <p>
      <b>
        Hello,
        <jsp:getProperty name="helper" property="firstName" />!
      </b>
    </p>
    <p>Here are your latest watch list price quotes:</p>
    <table columns="3">
      <helperTag:printData useLinks="no" currency="USD" />
    </table>
  </card>
</wml>

```

Prints the WML-version
without links ←

Listing 5.7 The British English WML watch list JSP

```

<jsp:useBean name="helper" scope="page"
    class="examples.chapter5.XMLHelperBean" />
<%@ taglib uri="example.tld" prefix="helperTag" %>

<wml>
  <card id="main" title="Your Watch List">
    <do type="accept" name="do-back" label="Back">
      <go href="http://www.exampleco.com/home.wml" />
    </do>
    <do type="accept" name="do-buy" label="Buy Shares">
      <go href="http://www.exampleco.com/buyShares" />
    </do>
    <do type="accept" name="do-sell" label="Sell Shares">
      <go href="http://www.exampleco.com/sellShares" />
    </do>
    <p><b>Greetings, Mr.
      <jsp:getProperty name="helper" property="lastName" />!
    </b></p>

```

```
<p>Here are the latest prices on your stocks of interest.</p>
<table columns="3">
  <helperTag:printData useLinks="no" currency="GBP" />
</table>
</card>
</wml>
```

5.2.4 Analyzing the results

We think you will admit that building a multidevice, multilocale presentation layer in J2EE is not easy. Imagine how much worse things could have been if we had chosen some less-complimentary output formats or wildly different locales. Also consider how much effort would be involved in extending our example code to accommodate more languages, locales, or device types.

This is the area in which an XML-based presentation layer can come to the rescue to some extent. While serving substantially different views of the same application is always challenging, some XML tools have recently emerged to make the process a bit easier for you. That is, after you learn to use them.

5.3 The J2EE/XML approach

The first XML architectural alternative we examine involves the combined use of XSLT with the J2EE presentation components. In chapter 2, we talked briefly about what XSLT is and how to use it via JAXP. You will recall that XSLT provides a general way to transform XML into virtually any output format. This comes in very handy when generating thin-client user interfaces like the one we have been working with in our example.

The output format of an XSLT process is determined by the transformation rules defined within an XSL stylesheet. In this section the desired output formats are HTML and WML. And to emphasize the capabilities of XSLT, we use it to generate PDFs from XML in this section too.

If you are still a bit fuzzy on XSLT concepts, you can learn more of the basics from the XSLT references in the bibliography or via an online tutorial. An excellent introduction to XSLT can be found online at <http://www.zvon.org>.

5.3.1 Adding XSLT to the web process flow

In section 5.2, we created a controller servlet, a custom tag/JavaBean component, and four JSPs to render the watch list page. Adding XSLT processing to the mix has the following impact on our design:

- The JSPs are no longer necessary.
- The custom tag/JavaBean component is no longer necessary.
- We require a new, outbound filter component to handle the XSLT process.
- We must modify our `WatchListJSPServlet` to remove the JSP forwarding code.

Filters are the latest addition to the J2EE presentation framework. They are useful for chaining together a web processing workflow. A filter can be applied to a specific type of request or globally across your entire application. Filters can perform preprocessing of a request (before it reaches the servlet) or post-processing. In our example case, we are interested in postprocessing any request that is handled by our Controller servlet.

The XSLT request handling flow now begins when our controller receives a request for a stock watch list. The servlet interacts with the application logic layer via the `ListBuilder` component, which still returns its result in the form of a `JDOM Document`. The view selection logic is now handled by our new filter component, which selects among XSL stylesheets instead of JSP pages.

This new architecture is an implementation of the Decorating Filter J2EE pattern, which you can learn more about in appendix A. Figure 5.6 depicts our implementation graphically.

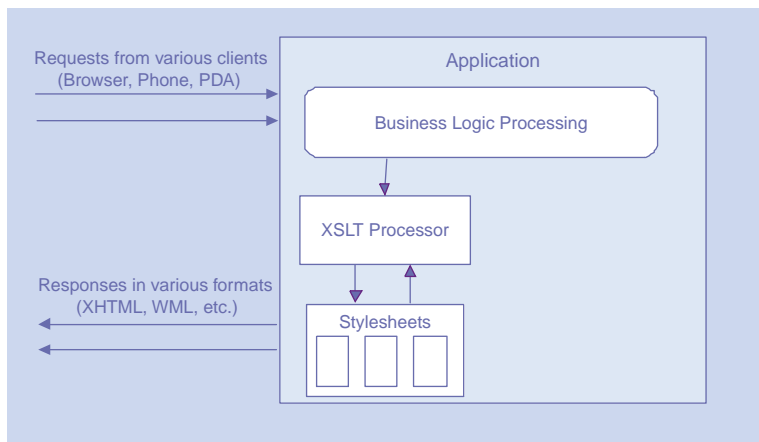


Figure 5.6 XSLT web request handling

The XSLT filtering process

We begin our XSLT example by developing the filter that will manage the stylesheet selection and XSLT transformation process. Here is an overview of how the filter works:

- 1 Each web request for the watch list page is intercepted by the filter and passed off to our controller servlet for processing.
- 2 The `JDOM Document` is returned to the filter via the `HttpRequest` object.
- 3 The filter determines the device type and preferred locale of the requesting client, just as our original servlet used to do.
- 4 The filter selects the most appropriate XSL stylesheet and invokes an XSLT processor to transform the JDOM results into a target output format
- 5 The filter sends the result of the XSLT transformation back to the user's device, where it is rendered for display.

For the sake of clarity, we oversimplify the view selection and XSLT transformation process. As discussed in chapter 2, special attention must be paid to precompiling and caching stylesheets in a production application, due to the performance characteristics of XSLT. In this version of our example, we concentrate on the interesting part, which is the transformation itself.

Modifying the servlet

Because our filter will select the appropriate stylesheet and no JSPs need to be invoked, our modified `WatchListServlet` component becomes quite simple. Its source code is shown in listing 5.8. The servlet now interacts with the `ListBuilder` interface and stores the `JDOM Document` in the `HttpRequest` object.

Listing 5.8 The modified watch list servlet

```
import org.jdom.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * The stock watchlist servlet with XSLT
 */
public class WatchListServlet extends HttpServlet {

    private ListBuilder builderInterface = new ListBuilder();
    private ServletConfig config;
    private ServletContext context;
```

```

public WatchListServlet() { super(); }

public void init(ServletConfig config)
    throws ServletException {
    this.config = config;
    this.context = config.getServletContext();
}

public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    HttpSession session = request.getSession(false);
    if (session == null) {
        context.getRequestDispatcher("/login.jsp")
            .forward(request, response);
        return;
    }
    String userId = (String) session.getAttribute("userId");
    Document quoteList = builderInterface.getWatchList(userId);
    request.setAttribute("quoteList", quoteList);
}

public void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
}

```

← No need to wrap the Document anymore

Building the filter

Our filter implementation is a class that implements the `javax.servlet.Filter` interface. The J2EE web container invokes the filter based on URL pattern matching, as it does with servlets. When a filter matching a specific URL pattern is found, the container invokes its `doFilter` method, the signature of which is as follows:

```

public void doFilter(ServletRequest request,
    ServletResponse response,
    FilterChain chain)
    throws IOException, ServletException;

```

The `FilterChain` parameter is a representation of all the processing to be done during this request, including the servlet(s), JSP(s), and other filter(s) that may be invoked. Since our filter is a postprocessing one, the first thing we do in our `doFilter` method is execute the `FilterChain`:

```
chain.doFilter(request, response);
```

This, in effect, calls the `WatchListServlet` via the web container, which places the `JDOM Document` in which we are interested into the request object. We then access the `Document` in the `doFilter` method.

```
HttpServletRequest httpRequest = (HttpServletRequest) request;  
Document outputDoc = (Document) httpRequest.getAttribute("quoteList");
```

Next, we call some helper methods to determine the appropriate stylesheet for the transformation.

```
String outputFormat = getOutputFormat(httpRequest);  
String locale = getLocaleString(httpRequest);  
String stylesheetPath = getStylesheet(outputFormat, locale);
```

You can see the bodies of these methods in listing 5.9. They are similar to those of our earlier `WatchListJSPServlet` component. Now that we have the XML document and know which stylesheet we want to use, we perform the transformation via the JAXP API.

```
TransformerFactory myFactory = TransformerFactory.newInstance();  
Transformer myTransformer = myFactory.newTransformer(new  
    StreamSource(stylesheetPath));  
JDOMResult result = new JDOMResult();  
myTransformer.transform( new JDOMSource( outputDoc ), result );
```

Now, all that is left to do is write the XSLT output back to the client, via the `HttpResponse` object.

```
Document resultDoc = result.getDocument();  
XMLOutputter xOut = new XMLOutputter();  
if (outputFormat.equals("wml"))  
    response.setContentType("text/vnd.wap.wml");  
PrintWriter out = response.getWriter();  
xOut.output( resultDoc, out );
```

Listing 5.9 provides the complete implementation of our `XSLTFilter` class.

Listing 5.9 XSLT filter code

```
import javax.servlet.*;  
import javax.servlet.http.*;  
import java.io.*;  
import java.util.*;  
import org.jdom.*;  
import org.jdom.output.*;  
import org.jdom.transform.*;  
import javax.xml.transform.*;  
import javax.xml.transform.stream.*;
```

```
public class XSLTFilter implements Filter {
    private FilterConfig filterConfig;

    public void init(FilterConfig filterConfig)
        throws ServletException {
        this.filterConfig = filterConfig;
    }

    public FilterConfig getFilterConfig() {
        return this.filterConfig;
    }

    public void setFilterConfig(FilterConfig filterConfig) {
        this.filterConfig = filterConfig;
    }

    public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain)
        throws IOException, ServletException {
        try {
            chain.doFilter(request, response);
            HttpServletRequest httpRequest
                = (HttpServletRequest) request;
            Document outputDoc
                = (Document) httpRequest.getAttribute("quoteList");
            if (outputDoc == null) return;

            String outputFormat = getOutputFormat(httpRequest);
            String locale = getLocaleString(httpRequest);
            String stylesheetPath
                = getStylesheet(outputFormat, locale);

            TransformerFactory myFactory
                = TransformerFactory.newInstance();
            Transformer myTransformer
                = myFactory.newTransformer(
                    new StreamSource(stylesheetPath));

            JDOMResult result = new JDOMResult();
            myTransformer.transform(
                new JDOMSource( outputDoc ), result );
            Document resultDoc = result.getDocument();
            XMLOutputter xOut = new XMLOutputter();
            if (outputFormat.equals("wml"))
                response.setContentType("text/vnd.wap.wml");
            PrintWriter out = response.getWriter();
            xOut.output( resultDoc, out );

        } catch (Exception e) {
            System.out.println("Error was:" + e.getMessage());
        }
    }
}
```

```
private String getOutputFormat(HttpServletRequest request) {
    String userAgent = request.getHeader("User-Agent");
    // this is where your robust user-agent lookup should happen
    if (userAgent.indexOf("UP.Browser") >= 0)
        return "wml";
    return "html";
}

private String getLocaleString(HttpServletRequest request) {
    Enumeration locales = request.getHeaders("Accept-Language");
    while (locales.hasMoreElements()) {
        String locale = (String) locales.nextElement();
        if (locale.equalsIgnoreCase("en_GB"))
            return "en_GB";
    }
    return "en_US";
}

private String getStylesheet(String outputFormat, String locale) {
    if (locale.equals("en_US")) {
        if (outputFormat.equals("html"))
            return "watchlist.html.en_US.xml";
        else
            return "watchlist.wml.en_US.xml";
    } else {
        if (outputFormat.equals("html"))
            return "watchlist.html.en_GB.xml";
        else
            return "watchlist.wml.en_GB.xml";
    }
}

public void destroy() {}
}
```

Developing the stylesheets

The final four pieces of our new, XSLT-enabled solution are the four XSL stylesheets used to transform the XML into output. We need to convert the JSPs, JavaBeans, and custom tag code developed in section 5.2 into four sets of XSLT transformation rules. Although there are a few different ways to develop an XSL stylesheet, the most straightforward is the template-based approach. XSL stylesheets developed in this manner most closely resemble the JSP templates with which you are already familiar.

Listing 5.10 contains the XSL stylesheet used to convert the quotes-list XML document into HTML format for U.S. users. Note the resemblance of this file to an HTML source file. The major differences are the wrapping of the entire document in an `<xsl:stylesheet>` element and a global `<xsl:template>`

element. The `<xsl:stylesheet>` element identifies this file as a set of transformation rules, and the `<xsl:template>` element is our global transformation rule that will be applied to the root node of the XML source document.

A thorough analysis of XSLT development is beyond the scope of this book. Note, however, the XML-based control structures (e.g., `<xsl:for-each>`) and variable substitution (e.g., `<xsl:value-of>`) that can be accomplished in these stylesheets. XSLT is a powerful tool for transforming XML in a variety of ways.

Listing 5.10 The U.S. English HTML-producing stylesheet

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

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:output method="html"
    doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
    />

  <xsl:template match="/">
    <html>
    <head><title>Your Watch List</title></head>
    <body>
    <h1>Your Stock Price Watch List</h1>

    <h3>
      Hello,
      <xsl:value-of select="/customer[@first-name]" />
    </h3>
    <h3>
      Here are the latest price quotes for
      your watch list stocks.
    </h3>
    <p><i>
      Price quotes were obtained at
      <xsl:value-of select="/quote-list[@time]" />
      on
      <xsl:value-of select="/quote-list[@date]" />
    </i></p>

    <table cellpadding="5" cellspacing="0" border="1">
    <tr>
      <th>Stock Symbol</th>
      <th>Company Name</th>
      <th>Last Price</th>
```

Allows us to use special HTML entities like `<`; and ` `;

The beginning of our only global transformation rule

Uses XPath to select customer first name

Gets the generation time from the quotes-list document

Gets the generation date from the quotes-list document

XSL and the developer can focus on the processing that generates the XML. These tasks can be performed relatively independently of each other.

The separation of roles advantage is not without its own challenges, however. For example, developing a user interface in XSL is far more difficult than using standard HTML and requires a programming skill-set that is uncommon among graphic designers. Graphical tools are expected to alleviate this problem somewhat in the future. Another challenge for the practicality of this architecture is the current lack of XSLT skill-sets in the industry. Though this will change over time, it has been an adoption hurdle for integrating XML into the presentation layer.

As we stated in chapter 2, performance will be the major factor in enabling the widespread deployment of XSLT-based presentation layers. It may produce an architecturally superior solution, but it must perform well to be adopted. All indications point toward a steady increase in XSLT performance as the technology matures.

5.3.3 *Extending to binary formats*

Before leaving the subject of interface rendering with XSLT, we should highlight the abilities of XSLT formatting objects. A popular requirement for advanced web applications today is to dynamically generate binary files that are more difficult or impossible to manipulate once generated.

DEFINITION *XSLT formatting objects (FO)* is a portion of the XSLT specification that defines the manner in which XML documents can be transformed into binary output via XSLT.

Using an implementation of XSLT formatting objects, you can transform your dynamic XML data into a binary format like PDF on the fly. To prove the point and see how it is done, we extend our watch list example to generate PDF files rather than HTML documents in this section.

For an implementation of XSL formatting objects, we turn once again to the Apache Software Foundation. ASF has a project called FOP that is currently a partial implementation of formatting objects. You can download the binaries, source, and documentation from <http://xml.apache.org/fop>.

For the sake of discussion, let us suppose that we need to output the stock watch list page in PDF format instead of HTML for web-based clients. Perhaps we have a fraud concern that someone might download the HTML source, modify it, and then claim that our pricing information was incorrect and cost

them money. Since it is difficult to steal and modify WML from a mobile device, we are only concerned with traditional, HTML web browsers.

Modifying our XSLT version of the example to generate PDF instead of HTML involves two steps:

- We must modify our HTML-producing stylesheets to produce an FOP formatting object tree instead of an HTML page.
- We must add a final step to the XSLT filter component to invoke the FOP API and convert the formatting object tree to a PDF document.

DEFINITION A *formatting object tree* is a specialized XSL stylesheet that contains print formatting instructions. The FOP `Driver` component uses these instructions to create a PDF file from an XML document.

The modified process flow for generating PDF instead of HTML is depicted in figure 5.7.

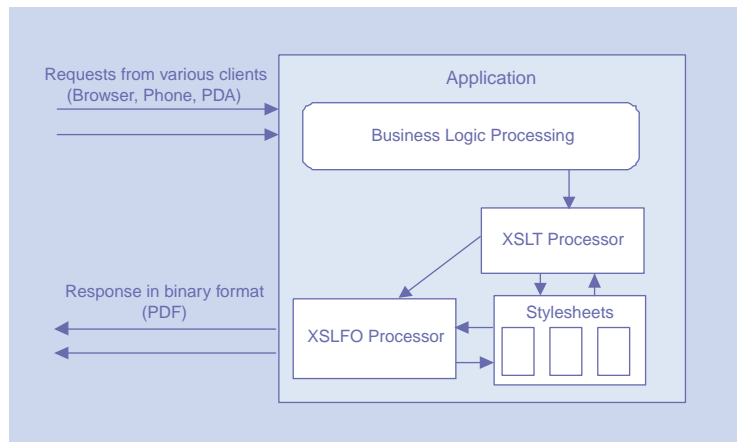


Figure 5.7 Dynamic PDF generation process flow

Creating a formatting tree

The basic structure of a formatting tree stylesheet is depicted in figure 5.8. The tree consists of two main components: layouts and page sequences. A *layout* describes a page template to be applied to one or more page sequences. Each *page sequence* defines the actual content that will appear in the PDF, including all of its formatting information. For our example, we create one layout (a master template)

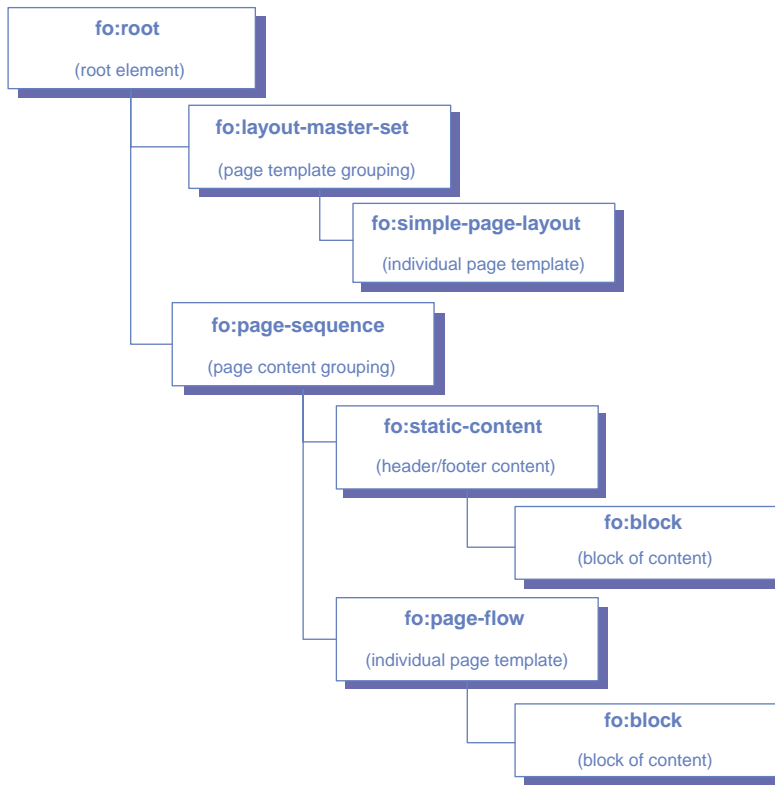


Figure 5.8 Logical structure of a FO formatting tree

and one page sequence. Our page sequence will contain a single page showing the same information as our HTML page did.

Listing 5.11 contains the complete formatting tree XSL stylesheet to produce the U.S. English version of the watch list page in PDF format.

Listing 5.11 Formatting tree for U.S. English stock watch list

```

<?xml version="1.0"?>

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  xmlns:fo="http://www.w3.org/1999/XSL/Format" <-- Identifies the XSLT
  > FO XML
  > namespace

<xsl:template match = "/">

```

```

<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <!-- define page layout -->
  <fo:layout-master-set>
    <fo:simple-page-master master-name="simple"
      page-height="29.7cm"
      page-width="21cm"
      margin-top="1.5cm"
      margin-bottom="2cm"
      margin-left="2.5cm"
      margin-right="2.5cm">
      <fo:region-body margin-top="3cm"/>
      <fo:region-before extent="1.5cm"/>
      <fo:region-after extent="1.5cm"/>
    </fo:simple-page-master>
  </fo:layout-master-set>

  <!-- define the content -->
  <fo:page-sequence master-name="simple">
    <fo:static-content flow-name="xsl-region-before">
      <fo:block text-align="end"
        font-size="10pt"
        font-family="serif"
        line-height="14pt" >
        Watch List - Customer
        <xsl:value-of
          select="./quote-list/customer/@id"/>
      </fo:block>
    </fo:static-content>

    <fo:flow flow-name="xsl-region-body">
      <fo:block font-size="16pt"
        font-family="sans-serif"
        font-weight="bold"
        line-height="26pt"
        space-after.optimum="12pt"
        background-color="blue"
        color="white"
        text-align="center">
        Your Stock Watch List
      </fo:block>
      <fo:block font-size="12pt"
        font-family="sans-serif"
        font-weight="bold"
        line-height="18pt"
        space-after.optimum="10pt"
        start-indent="10pt">
        Hello,
        <xsl:value-of

```

Defines a page sequence, bound to our template above

Defines a global page header for the sequence

Prints customer id from the data document

Defines document contents

```

        select="./quote-list/customer/@first-name"/>
</fo:block>
<fo:block font-size="10pt"
  font-family="sans-serif"
  font-style="italic"
  line-height="18pt"
  space-after.optimum="10pt"
  start-indent="15pt">
  Prices were obtained at
    <xsl:value-of select="./quote-list/@time"/>
    on
    <xsl:value-of select="./quote-list/@date"/>
</fo:block>
<fo:table>
  <fo:table-column column-width="3cm"/>
  <fo:table-column column-width="7cm"/>
  <fo:table-column column-width="3cm"/>
<fo:table-header font-size="10pt"
  line-height="14pt"
  font-family="sans-serif">
<fo:table-row font-weight="bold">
  <fo:table-cell text-align="start">
    <fo:block>SYMBOL</fo:block>
  </fo:table-cell>
  <fo:table-cell text-align="start">
    <fo:block>COMPANY NAME</fo:block>
  </fo:table-cell>
  <fo:table-cell text-align="start">
    <fo:block>SHARE PRICE</fo:block>
  </fo:table-cell>
</fo:table-row>
</fo:table-header>
  <fo:table-body font-size="10pt"
  line-height="16pt"
  font-family="sans-serif">
    <xsl:for-each select="//quote">
      <fo:table-row>
        <fo:table-cell>
          <fo:block text-align="start" >
            <xsl:value-of select="@symbol"/>
          </fo:block>
        </fo:table-cell>
        <fo:table-cell>
          <fo:block text-align="start" >
            <xsl:value-of select="@name"/>
          </fo:block>
        </fo:table-cell>
        <fo:table-cell>
          <fo:block text-align="start" >

```

← Constructs the watch list table, just as we did in textual formats

```

        $ <xsl:value-of
            select="./price[@currency='USD']/@amount"/>
        </fo:block>
    </fo:table-cell>
</fo:table-row>
</xsl:for-each>
</fo:table-body>
</fo:table>
</fo:flow>
</fo:page-sequence>
</fo:root>
</xsl:template>
</xsl:stylesheet>

```

To make use of the FO stylesheet, we must invoke the Apache FOP API. Since we do not want to tie our filter implementation to a specific FO implementation, we chose to wrap the use of FOP with an adapter object called `PDFWriter`. This component takes a formatting tree stylesheet path and an XML input source and writes the PDF to a specified output stream. To do its work, the `PDFWriter` uses both Apache FOP and the JAXP API for XSLT. The code for this adapter is given in listing 5.12.

Listing 5.12 An adapter for Apache FOP

```

import java.io.*;

import org.xml.sax.InputSource;
import org.apache.fop.apps.Driver;
import org.apache.fop.apps.Version;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;

public class PDFWriter {
    protected Transformer transformer = null;

    public PDFWriter(StreamSource source)
        throws TransformerConfigurationException {
        TransformerFactory factory
            = TransformerFactory.newInstance();
        transformer = factory.newTransformer(source);
    }

    public PDFWriter(String xslFilePath)
        throws TransformerConfigurationException,
            FileNotFoundException {
        this(new StreamSource(new FileInputStream(xslFilePath)));
    }
}

```

Uses JAXP to transform the XML data into a FO tree

```

protected byte[] invokeFOP(InputSource foSource)
    throws Exception {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    Driver driver = new Driver(foSource, out);
    driver.run();
    return out.toByteArray();
}

public byte[] generatePDF(StreamSource xmlSource)
    throws Exception {
    ByteArrayOutputStream baos
        = new ByteArrayOutputStream();
    StreamResult foResult = new StreamResult(baos);
    transformer.transform(xmlSource, foResult);
    ByteArrayInputStream bais
        = new ByteArrayInputStream( baos.toByteArray() );
    return invokeFOP( new InputSource(bais) );
}

public static void createPDFFromXML(String xslFilePath,
    InputStream xmlIn,
    OutputStream pdfOut)

    throws Exception {
    PDFWriter writer = new PDFWriter(xslFilePath);
    byte[] PDFbytes
        = writer.generatePDF( new StreamSource(xmlIn) );
    pdfOut.write(PDFbytes, 0, PDFbytes.length);
}
}

```

Defines a set of page templates containing a single page named simple

Transforms XML to PDF and writes to an output stream

The last thing we need to do is modify our XSLT filter to use the `PDFWriter` when `html` is the output format. The modified `XSLTPDFFilter` class is shown in listing 5.13.

Listing 5.13 The modified XSLT filter class

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import org.jdom.*;
import org.jdom.output.*;
import org.jdom.transform.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;

public class XSLTFilter implements Filter {

    private FilterConfig filterConfig;

```

```

public void init(FilterConfig filterConfig)
    throws ServletException {
    this.filterConfig = filterConfig;
}

public FilterConfig getFilterConfig() {
    return this.filterConfig;
}

public void setFilterConfig(FilterConfig filterConfig) {
    this.filterConfig = filterConfig;
}

public void doFilter(ServletRequest request,
                    ServletResponse response,
                    FilterChain chain)
    throws IOException, ServletException {
    try {
        chain.doFilter(request, response);
        HttpServletRequest httpRequest
            = (HttpServletRequest) request;
        Document outputDoc
            = (Document) httpRequest.getAttribute("quoteList");
        if (outputDoc == null) return;

        String outputFormat = getOutputFormat(httpRequest);
        String locale = getLocaleString(httpRequest);
        String stylesheetPath
            = getStylesheet(outputFormat, locale);

        XMLOutputter xOut = new XMLOutputter();
        if (outputFormat.equals("html")) {
            ByteArrayOutputStream baos
                = new ByteArrayOutputStream();
            xOut.output(outputDoc, baos);
            ByteArrayInputStream bais
                = new ByteArrayInputStream( baos.toByteArray() );
            // get the response output stream
            response.setContentType("application/pdf");
            OutputStream out = response.getOutputStream();
            PDFWriter.createPDFFromXML( stylesheetPath, bais, out);
        }
        out.close();
    } else {
        // wml format
        TransformerFactory myFactory
            = TransformerFactory.newInstance();
        Transformer myTransformer
            = myFactory.newTransformer(
                new StreamSource(stylesheetPath));
        JDOMResult result = new JDOMResult();
        myTransformer.transform(
            new JDOMSource( outputDoc ), result );
    }
}

```

Converts JDOM to an
input stream

Creates and
saves PDF


```

        Document resultDoc = result.getDocument();
        response.setContentType("text/vnd.wap.wml");
        PrintWriter out = response.getWriter();
        xOut.output( resultDoc, out );
    }
} catch (Exception e) {
    System.out.println("Error was:" + e.getMessage());
}
}
private String getOutputFormat(HttpServletRequest request) {
    String userAgent = request.getHeader("User-Agent");
    if (userAgent.indexOf("UP.Browser") >= 0)
        return "wml";
    return "html";
}

private String getLocaleString(HttpServletRequest request) {
    Enumeration locales = request.getHeaders("Accept-Language");
    while (locales.hasMoreElements()) {
        String locale = (String) locales.nextElement();
        if (locale.equalsIgnoreCase("en_GB"))
            return "en_GB";
    }
    return "en_US";
}

private String getStylesheet(String outputFormat, String locale) {
    if (locale.equals("en_US")) {
        if (outputFormat.equals("html"))
            return "watchlist.pdf.en_US.xsl";
        else
            return "watchlist.wml.en_US.xsl";
    } else {
        if (outputFormat.equals("html"))
            return "watchlist.pdf.en_GB.xsl";
        else
            return "watchlist.wml.en_GB.xsl";
    }
}

}
public void destroy() {}
}

```

Uses our FO stylesheets
instead of the HTML-
producing ones

Figure 5.9 shows the fruit of our labor, a dynamically generated PDF containing our stock watch list data.



SYMBOL	COMPANY NAME	SHARE PRICE
SRMC	Sierra Monitor Corporation	\$ 2.00
IBM	International Business Machines	\$ 135.00
ORCL	Oracle Corporation	\$ 15.00

Figure 5.9 The U.S. English PDF version of the stock watch list

5.4 XML web publishing frameworks

The architecture presented in the previous section requires custom code development that can be difficult and time-consuming. In this section, we explore a possible alternative to the custom integration work called web publishing frameworks.

DEFINITION A *web publishing framework* is a software suite design to speed the development of an XML-based presentation layer.

Web publishing frameworks combine Java and XML technologies into a cohesive and usable architecture. They are an out-of-the-box solution for generating your user interface. Using a web publishing framework, you can create a production-ready, XML-based presentation layer without having to write custom code that integrates XML into your architecture. These frameworks were built to solve the same challenge that we outlined earlier in this chapter—producing various views of the same content while maintaining a separation between presentation logic and style.

Web publishing frameworks that are based on XML technologies are still relatively new. Their reliability depends on the stability of underlying components, including the XML parser and XSLT processor. A few popular web publishing framework products include the following:

- Webmacro (<http://www.webmacro.org>)
- Enhydra (<http://www.enhydra.org>)
- Cocoon (<http://xml.apache.org/cocoon>)

For purposes of comparison with the XSLT approach from the previous section, we now explore how our watch list example could be developed within the Cocoon web publishing framework.

5.4.1 Introduction to Cocoon architecture

At the time of this writing, the Apache Software Foundation's Cocoon is one of the most stable and feature-rich XML web publishing frameworks. User interface development with Cocoon involves the creation of XSL stylesheets and XSP pages. Since XSP is a technology currently limited to Cocoon, we will concentrate on the more generic, XSLT capabilities of Cocoon in this section. Figure 5.10 depicts the Cocoon processing flow.

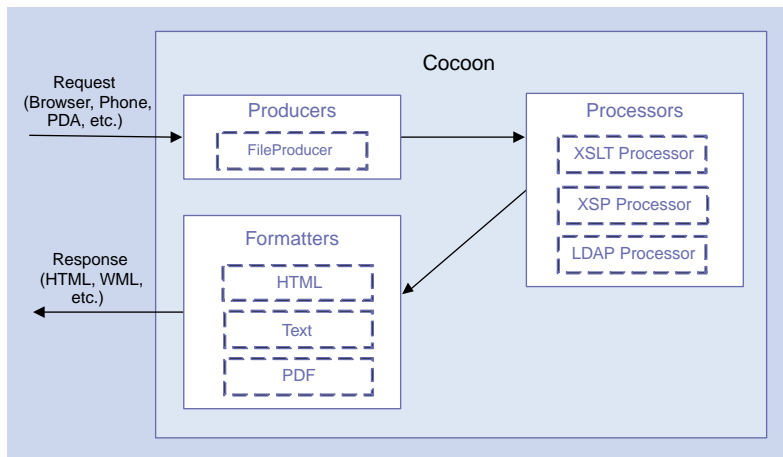


Figure 5.10 The Cocoon request processing flow

DEFINITION *XML Server Pages (XSP)* is a working draft specification for an XML-based program generation language. XSPs contain directives that control how a given XML data set is processed.

The simplest way to use Cocoon is to add special processing instructions to your XML data documents. These processing instructions allow Cocoon to process and format your data and deliver it to a requesting client. Supported output formats include WML, PDF, XML, and XHTML.

Cocoon producers

Producers are software components responsible for generating XML data. They are the equivalent of a servlet in that they receive and process an `HttpServletRequest`. This is just one of the areas in which Cocoon is extensible. You can implement your own producers to perform custom processing. Cocoon ships with a `FileProducer` that loads a requested file from the file system.

Cocoon processors

Once the data has been produced, it is available for processing. A *processor* is a component responsible for performing an operation such as an XSLT transformation on the XML data generated by a producer. Cocoon contains the following processors out-of-the-box:

- An XSLT processor
- An LDAP processor
- An XSP processor

Writing your own processor is similar to writing a JSP custom tag. The tag is created, associated with some behavior, and included in a page.

Cocoon formatters

Formatters are helper components that may be applied to a response before it is returned to the requesting client. Formatters are used to wrap output content with additional formatting information. Formatters do things such as placing tags around such markup content as HTML documents and creating a final PDF from a XSLFO formatting tree.

5.4.2 Using Cocoon to render the watch list page

Let us put Cocoon to work on our example XML document. We will use the standard Cocoon XSLT processor to perform an XSLT transformation on our quote-list XML. Fortunately for us, we already developed the XSL required to make this work in section 5.2.

The only modification we need to make to the XML document returned from the application logic layer (the `ListBuilder` interface in the example) is to

add a Cocoon processing instruction and references to our XSL stylesheets within the XML data document. The Cocoon directive is as follows:

```
<?cocoon-process type=xslt ?>
```

Then we add two processing instructions that describe the HTML and WML stylesheets to be applied to the data. For the U.S. locale, the instructions are as follows:

```
<?xml-stylesheet href=watchlist.html.en_US.xsl type=text/xsl ?>  
<?xml-stylesheet href=watchlist.wml.en_US.xsl type=text/xsl  
  media=wap ?>
```

The `media=wap` attribute in the second processing instruction tells Cocoon to select this stylesheet for WML-based clients. In other cases, the default stylesheet will be used.

Cocoon is designed to be accessed as a servlet, but can be invoked via an API call as well. In listing 5.14, our modified `WatchListServlet` adds the appropriate processing instructions to the XML data document and invokes the Cocoon engine to perform the transformation and delivery back to the client.

Listing 5.14 Invoking Cocoon from the watch list servlet

```
import org.jdom.*;  
  
import java.io.*;  
import java.util.*;  
import javax.servlet.*;  
import javax.servlet.http.*;  
  
import org.apache.cocoon.Engine;  
import org.apache.cocoon.util.CocoonServletRequest;  
  
public class WatchListServletWithCocoon extends HttpServlet {  
  
    private ListBuilder builderInterface = new ListBuilder();  
    private ServletConfig config;  
    private ServletContext context;  
  
    public WatchListServletWithCocoon() { super(); }  
  
    public void init(ServletConfig config)  
        throws ServletException {  
        this.config = config;  
        this.context = config.getServletContext();  
    }  
  
    public void doGet(HttpServletRequest request,  
        HttpServletResponse response)  
        throws ServletException, IOException {
```

```

// get userid from HttpSession
HttpSession session = request.getSession(false);
if (session == null) {
    context.getRequestDispatcher("/login.jsp")
        .forward(request, response);
    return;
}
String userId = (String) session.getAttribute("userId");
Document quoteList =
    builderInterface.getWatchList(userId);

String localeString = getLocaleString(request);
String document
    = getOutputDocWithProcessingInstructions(quoteList,
                                             localeString);

try {
    Engine cocoonEngine = Engine.getInstance();
    CocoonServletRequest myReq
        = new CocoonServletRequest(document, request);
    cocoonEngine.handle(myReq, response);
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
}

private String getLocaleString(HttpServletRequest request) {
    Enumeration locales
        = request.getHeaders("Accept-Language");
    while (locales.hasMoreElements()) {
        String locale = (String) locales.nextElement();
        if (locale.equalsIgnoreCase("en_GB"))
            return "en_GB";
    }
    return "en_US";
}

private String
    getOutputDocWithProcessingInstructions(Document document,
                                           String locale) {
    if (locale.equals("en_US")) {
        document.addContent(
            new ProcessingInstruction("xml-stylesheet",
                "href=\"watchlist.html.en_US.xsl\" type=\"text/xsl\""));
        document.addContent(
            new ProcessingInstruction("xml-stylesheet",
                "href=\"watchlist.wml.en_US.xsl\" type=\"text/xsl\"
                media=\"wap\""));
    } else {
        document.addContent(
            new ProcessingInstruction("xml-stylesheet",
                "href=\"watchlist.html.en_GB.xsl\" type=\"text/xsl\""));
    }
}

```

Adds processing instructions for either locale

Obtains a handle to the Cocoon Engine

Wraps document as an HttpRequest

Invokes Cocoon to perform the XSLT transformation

Adds processing instructions for specified locale

```

        document.addContent(
            new ProcessingInstruction("xml-stylesheet",
                "href=\"watchlist.wml.en_GB.xml\" type=\"text/xsl\"
                media=\"wap\""));
    }
    return document.toString();
}

public void doPost(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
}

```

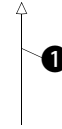


Figure 5.11 depicts our new presentation layer that combines Cocoon and the J2EE presentation layer components.

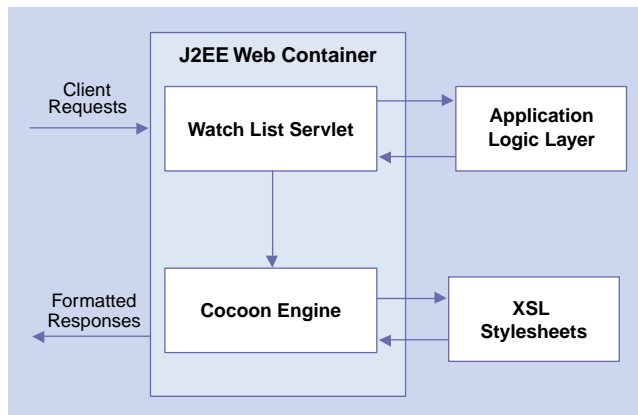


Figure 5.11
The J2EE Cocoon
presentation layer

5.4.3 Analyzing the results

The primary advantage of using a web publishing framework like Cocoon is the avoidance of writing a significant amount of Java code, which would otherwise be necessary to make the XSLT transformation process happen. From a development standpoint, you need only supply XML documents and XSL stylesheets to use the Cocoon framework. Web publishing frameworks also feature caching algorithms to optimize performance.

This convenience does come with a price. Despite precompiling stylesheets and caching at various levels, XSLT transformation does not perform as well as compiled templates, no matter what framework you use. Additionally, the

J2EE presentation framework is fast, reliable, and adequately satisfies most user interface needs.

This begs the question—should you replace your JSP infrastructure with XSLT, XSP, and Cocoon? At this point, we believe it is advisable to only augment your J2EE infrastructure with Cocoon and use XSP sparingly. Cocoon technology has a lot of potential, but is not yet mature enough to warrant redesigning your user interface. For now, the use of web publishing frameworks should be limited to the portion of your application that has advanced interface requirements like those described in this chapter.

5.5 *A word about client-side XSLT*

The architectures explored in this chapter perform XSLT transformations focused exclusively on the server side. This is because most J2EE applications use the thin-client model, wherein the server is required to do the work. Most web browsers still do not support client-side XSLT or they lack the processing power to perform transformations (e.g., wireless device browsers). This may change in the future, and a model in which the client side performs the transformation might become possible.

One of the potential advantages of the client-side approach is the relief of a heavy burden from the server-side, leading to faster responses and higher application throughput. Another advantage may be the ability to perform specialty processing such as conversion to voice or Braille. Client-side processing could be an integrated feature of the browser or implemented using applets. Currently, Internet Explorer 5 is the only popular browser with any real XML support.

5.6 *Summary*

This chapter was all about creating a more robust, flexible user interface using XML within the J2EE framework. We began by reviewing the common challenges involved in the development of a thin-client user interface to serve multiple types of users connecting via various types of devices. We detailed the difficulties involved in creating and maintaining a multilocale, multidevice presentation layer using J2EE alone. We then discussed two alternatives to the pure J2EE approach made available by the advent of XML technology.

The first alternative is to add XSLT processing to your J2EE request handling process. The recent addition of filters to the servlet API makes adding

XSLT much easier using the Decorating Filter pattern. Still, you have to roll your own code to take this approach.

The second alternative is to use XSLT via a web publishing framework like Apache Cocoon. This approach allows you to avoid custom coding of the XSLT extensions to your presentation layer and offers some potential performance benefits from caching algorithms and other means. XSLT is much slower than using compiled templates, but can be a reasonable alternative for large, complicated interfaces that have advanced requirements such as multilo-cale and/or multidevice support.

If you decide that a hybrid J2EE/XML approach is not right for your application, you should investigate other pure J2EE presentation frameworks to give your user interface development efforts a jump start. Two such popular frameworks are Struts and Velocity. Information on both can be found at <http://jakarta.apache.org>.

This chapter concludes our detailed discussion of using XML at each layer of your n-tier J2EE application. The next and final chapter summarizes the concepts and technologies from the first five chapters in the context of a cohesive case study.



Case study

This chapter

- Synthesizes concepts from previous chapters
- Exercises J2EE/XML hybrid architecture
- Presents an end-to-end example

Throughout this book, we have examined various technologies relating to the integration of J2EE and XML. Our goal has been to cover each component with sufficient breadth and depth to give you a solid understanding. Each individual component, however, is extremely complex and requires its own book for an exhaustive discussion. The goal of this chapter is to provide cohesion to our discussion and the topics that were previously explored. We use a case study to examine an end-to-end architecture that highlights J2EE and XML design patterns and technologies. Throughout this chapter, we produce an application and step through its implementation. All of the materials necessary to run the application can be found at <http://www.manning.com/gabrick/>.

6.1 *Case study requirements*

Our case study surrounds a fictional computer repair company named RepairCo. In addition to repairing computers that have been brought or shipped to their store locations, RepairCo sends certified technicians to customer sites for onsite troubleshooting and repair. RepairCo technicians, while certified and well trained, need access to the latest information on each machine that they fix. The application that we develop throughout this chapter enables real-time access to manuals, diagnostic material, and repair history. Our application is accessed through a handheld device with wireless Internet access.

The first step in our application development is the requirements process. Given the small scope of this implementation, we document our requirements informally in a small set of use cases. A heavy discussion about formal development methodologies such as the Rational Unified Process or eXtreme Programming is beyond the scope of this book. In an effort to limit the scope, we will specify three use cases that our application must satisfy:

- Use case 1: Viewing the application menu
- Use case 2: Viewing detailed machine information
- Use case 3: Viewing manufacturer information

The use cases are outlined in table 6.1.

Table 6.1 Use cases for the RepairCo application case study

Use case 1—User views application menu	
Name	1. User views application menu
Objective	User views application menu

(continued on next page)

Table 6.1 Use cases for the RepairCo application case study (continued)

Steps	Using handheld device, user launches WAP browser. User enters URL or selects bookmark for application home page. System presents application menu. Diagnose Application Hardware Problem View Availability of Replacement Systems Order Replacement Parts View Common System Problems
Use case 2—User views detailed machine information	
Name	2. User views detailed machine information
Objective	User views common problems for a specific machine
Steps	User selects VIEW COMMON PROBLEMS from the application menu. System presents a list of machines. Omnic 400 Amaya Workstation Computech ATX ACME xPad User selects the Amaya Workstation. System presents a list of common problems associated with that machine.
Use case 3—User views latest information from manufacturer	
Name	3. User views latest information from manufacturer
Objective	User views latest list of common problems.
Steps	User selects VIEW COMMON PROBLEMS from the application menu. System presents a list of machines. User selects Amaya Workstation. System presents a list of common problems associated with that machine. User selects update. System retrieves latest information from manufacturer and presents updated list of issues.

Given very minor modifications to the use cases, this could represent any thin-client application that is accessed over the Internet. The only piece of functionality that may not currently be very common is real-time access to partner information (use case #3). While this functionality may be very desirable, it has typically involved custom integration work with each partner. As we alluded in chapter 4, the growing popularity of web services will make real-time access to partner data a more reasonable undertaking. The lack of complexity in our requirements was intentional. The interesting points of our application lie in the implementation details. The more generic the requirements, the easier it is for you to apply the same design principles to your own applications.

6.2 The application environment

Before we examine the architecture of our application, we must review the application environment. There are myriad choices available for web servers, J2EE servers, and SOAP servers. Though we did choose software for the implementation of our case study, it should be clear that our intention is not to perform a software evaluation. On the server side, we chose BEA's WebLogic 6.1. It serves as our web server, J2EE container, and SOAP server. The focus of our discussion is the design and implementation of application, not the operating environment. With a few modifications, this application will run using different software products. For this reason, we note areas in our code that are server dependant.

For client-side access to our application, we use a Palm organizer with wireless Internet access. The technicians at RepairCo merely launch an Internet browser and select a bookmark to hit our application. The browser that we use is the EzWap browser from EZOS, which interprets and renders WML on

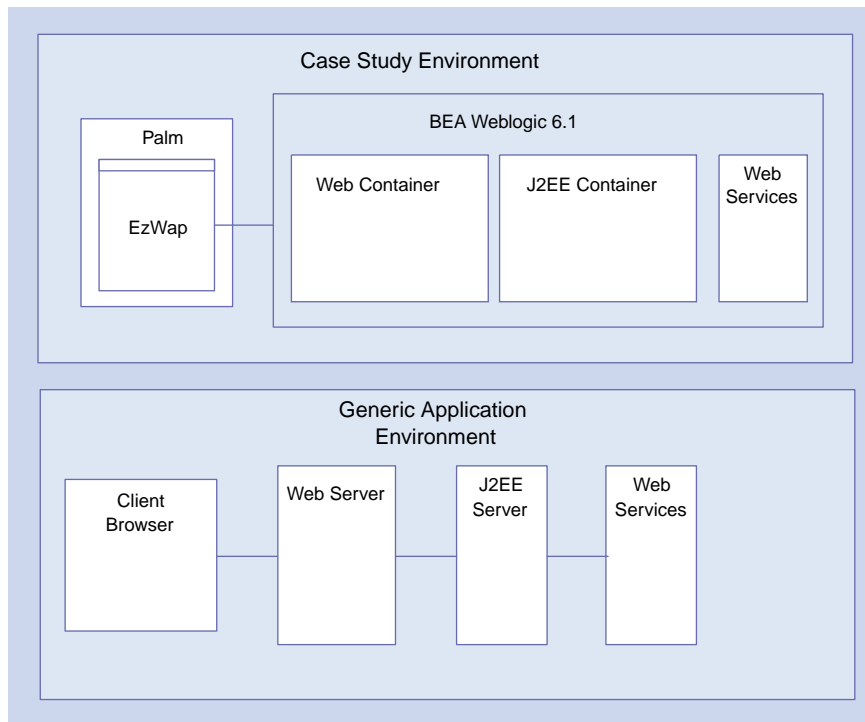


Figure 6.1 Case study application environment and generic application environment

the Palm device. Just as in the case of the server-side software, these components can be substituted with other software that renders WML. Figure 6.1 depicts the case study application environment and the generic components that can be used in their place.

6.3 The analysis phase

The next step in our case study is the analysis of our requirements and the development and refinement of our analysis models. In chapter 4, section 4.1 we discussed constraint-based modeling as a means of designing our system. In this vein, we begin our analysis with the services and data layer, because it is clearly the most complex portion of our implementation.

6.3.1 Services and data layer analysis

The first component we look at is the integration to our partner companies. For our purposes, we will develop the integration to only one of the manufacturers. Fortunately, our partner Amaya Inc. has a web service that we can integrate to provide support information for the Amaya Workstation. Their specifications for the RPC-style web service come in the form of a WSDL file. In order to get the latest support information, we use a SOAP server to connect to the web service and update our list. This process is a set of business logic that we encapsulate in a stateless session bean called `BugAccessorBean`. Our bean calls the web service by passing the name of the machine that we are repairing, and the web service returns an XML DOM containing the information that we requested. Figure 6.2 depicts this web service scenario. In order to implement our application, we do not need intimate knowledge about how Amaya implemented their web service. However, in the spirit of this case study, we will examine the Amaya implementation of the web service in section 6.8 so both sides of the implementation are clear.

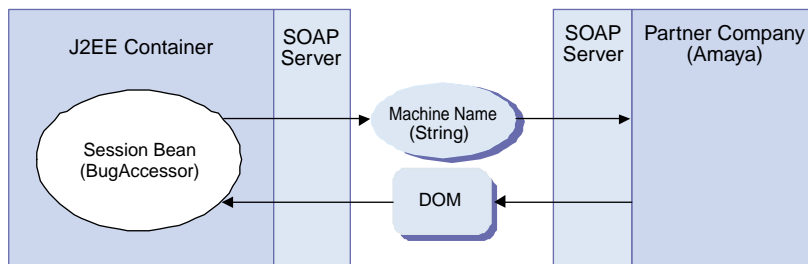


Figure 6.2 Connecting to partner company using a web service

6.3.2 Data storage analysis

The next choice that we need to make is a data storage mechanism. In chapter 3, we reviewed the advantages and disadvantages of several XML data storage options, such as XML databases, PDOM, and relational databases. In this case, we will use file system storage for our XML data so that we can open the file and examine its contents during processing. To satisfy our use cases, we must store the following data in table 6.2. These XML files will be reviewed in section 6.7.

Table 6.2 Case study data files

Purpose	Filename
Menu of functions available in our application	menu.xml
List of common problems associated with the Amaya machine	amaya.xml

Once again, for the purposes of our example, we will discuss integration with our partner Amaya only. Figure 6.3 displays an updated diagram of our architecture.

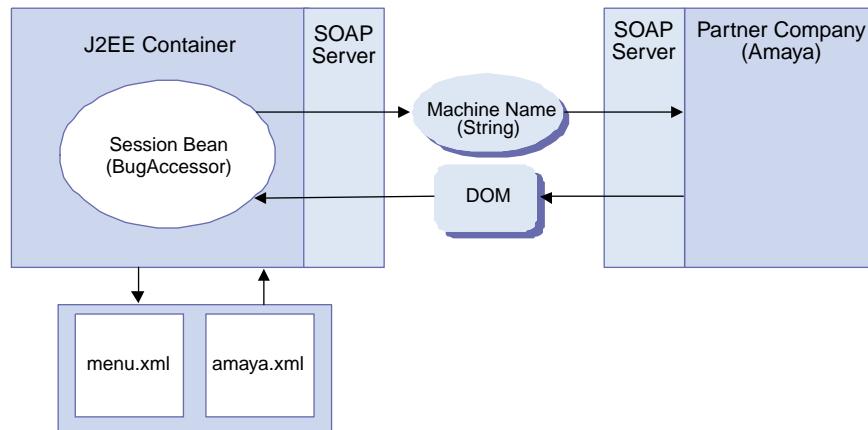


Figure 6.3 Data and services layer architecture

6.3.3 Other necessary components

While the `BugAccessor` bean may be responsible for updating our bug lists, we need several additional components to round out our architecture. First, we add a component called the `ApplicationMenu`. As implied by its name, the

ApplicationMenu is responsible for the initial actions that our application supports. It provides access to the `menu.xml` file. It also has an administrative interface so that RepairCo administrators can update the application.

In addition, we introduce a presentation mechanism and a control point for the web application. A servlet, named `DiagnosticApp`, will route all requests to the appropriate components. Once the business logic processing is complete, an `XSLTFilter` component will be responsible for transforming the content before it is rendered to the user. This presentation mechanism was discussed in chapter 5 and is an implementation of the Model-View-Controller pattern. It is a solid choice because it is very likely that there will be a future requirement for our application to support multiple output formats or languages. The flexibility of our filter enables this functionality to be added through XSLT transformation without retouching the business logic components. In some cases, this presentation framework may not be desirable because the XSLT transformation impacts performance. In this case, performance is not a primary concern because of the limited volume of requests that we receive from our technicians.

Figure 6.4 shows our complete analysis architecture. Though we will add design-level components in the next section, we can satisfy the required functionality with the tools shown here.

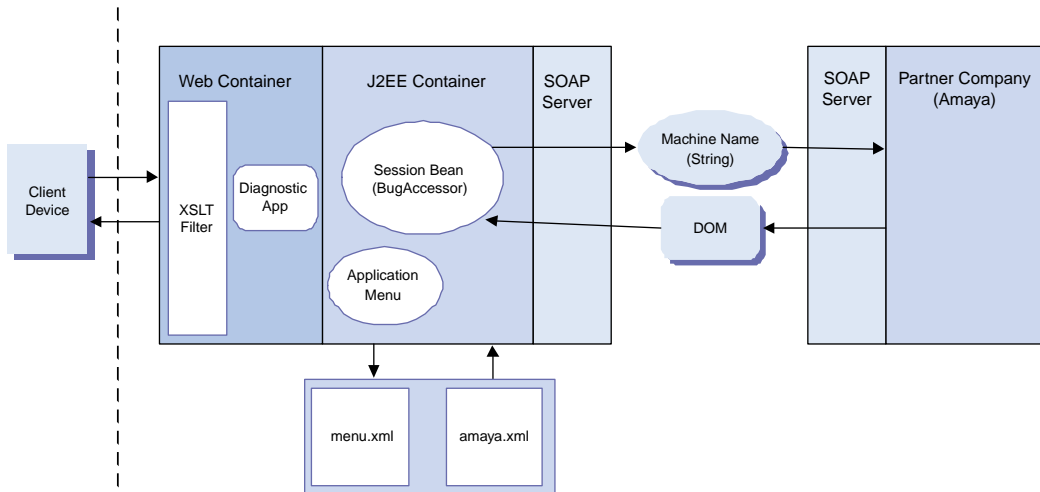


Figure 6.4 Completed case study analysis architecture

6.4 *The design phase*

Now that we have completed our requirement-gathering process and developed the analysis architecture, we move into design. During this phase, we detail each component and add any necessary implementation classes to our application. Also, we apply the principles and design patterns discussed throughout this book to create a more robust architecture.

6.4.1 *Designing the application logic layer*

The first area of the application that requires our attention is the application logic layer, the brains of our system. We must design the `BugAccessorBean` and related components before concerning ourselves with other issues such as user interface design.

The `BugAccessorBean` and `ComponentLocator`

The `BugAccessorBean` is a stateless session bean that is responsible for updating our bug list. In order to accomplish this, it communicates with our partner companies through a web service to retrieve the latest information. Our bean will have one method that contains the business logic called `updateBugList` that takes the name of the particular machine that is to be updated and returns a JDOM `Document` with the updated list of issues.

The Amaya web service is not the only service that our application must locate. The other components in our application also require naming services to locate our `BugAccessorBean`. For this reason, we add a design level class to our application called `ComponentLocator`. It will handle the logic surrounding locating components shielding the other classes from the complexity and removing potential redundant code. This class is an implementation of the Service Locator pattern. Additionally, we implement this class as a singleton using the Singleton pattern. The Singleton pattern uses one, and only one, instance of the class in a particular JVM. One reason for this design is that we save resources by creating the naming context only once. Also, this eliminates the remote calls necessary for each client to locate the `ComponentLocator` class. Lastly, as a service locator class, the `ComponentLocator` is not responsible for maintaining or writing any data. This enables our application to scale to multiple servers without risking inconsistencies in the state. For more information on the Service Locator pattern or the Singleton pattern, please see appendix A.

Figure 6.5 depicts these components in a class diagram using the Universal Modeling Language (UML). If you are unfamiliar with UML notation, there is a legend next to the `ComponentLocator` class that explains the composition of the diagram.

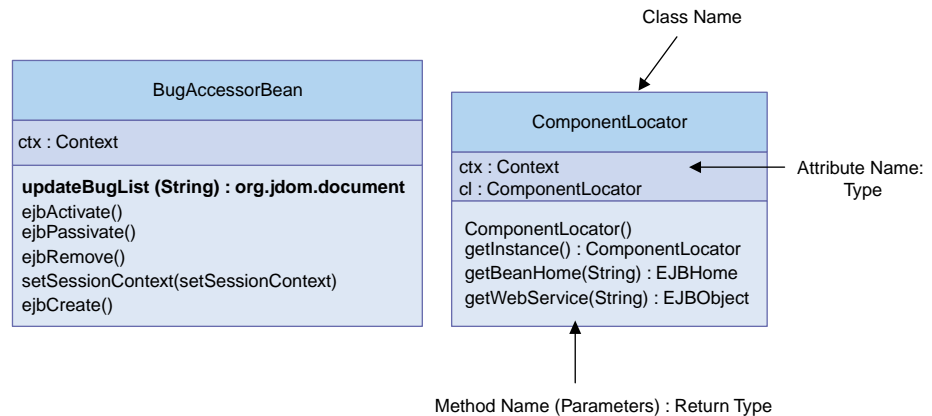


Figure 6.5 Case study design architecture: `BugAccessorBean` and `ComponentLocator`

The `ApplicationMenu` component

The `ApplicationMenu` is a class that is responsible for building the initial menu for our system. The information necessary to build the menu is stored in a file named `menu.xml`. This was designed intentionally, so that new branches and functions can be added to the system with relative ease. Each time a user request is received, it is impractical to reload this information from a file. This will undoubtedly cause performance problems. In order to alleviate this problem, we employ the Singleton pattern once again and load the data into our `ApplicationMenu` class. Once we have initially loaded the menu, the `getMenu` method merely returns the JDOM containing the menu information to the caller. This is a caching method often used when the data is accessed frequently and is not frequently modified. The reason that we have chosen this architecture over using a bean is that the remote calls necessary to look up a bean require too much overhead. One instance of this class in each JVM serves our needs. In a different situation in which our class performed more intense logic, we would reevaluate our design.

The next issue we must consider is the process for updating the menu. Unlike the `ComponentLocator` class, the `ApplicationMenu` must refresh itself each time the menu is updated. The solution to this problem is the use of the Service Activator pattern discussed in appendix A. In this pattern, a client subscribes to a JMS queue to receive asynchronous messages. The client listens for a message and performs some logic when a message is received. In our case, we configure a JMS queue and add the `ApplicationMenu` as a subscriber. Any

time the `menu.xml` file is updated, we send a message that causes the `ApplicationMenu` to refresh itself. Figure 6.6 depicts the administrative interface to the `ApplicationMenu`.

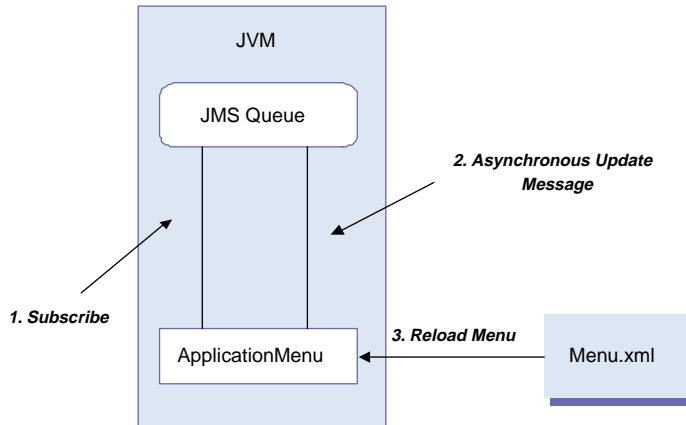


Figure 6.6 Administrative interface to `ApplicationMenu`

6.4.2 Designing the user interface

The final two components in our architecture are the presentation components. In chapter 5, we introduced the combination of a filter, a servlet, and a bean as an implementation of the Model-View-Controller pattern. In this paradigm, the servlet acts as the controller receiving requests and choosing the appropriate combination of data (model) and presentation (view) to render to the caller. The bean acts as the model and is only concerned with data. The filter acts as the view and renders the data in the appropriate format. The flexibility of this paradigm allows us to implement a J2EE architecture that can easily handle multiple presentation formats with minimal impact to the underlying data.

In our case study design, the `BugAccessorBean` and the `ApplicationMenu` act as the model. The `DiagnosticApp` is our controller servlet and the `XSLT-Filter` is our view. Our current requirements indicate that we must produce only WML, but this presentation framework is flexible enough to accommodate additional presentation formats if necessary. Figure 6.7 is the complete class diagram for our application.

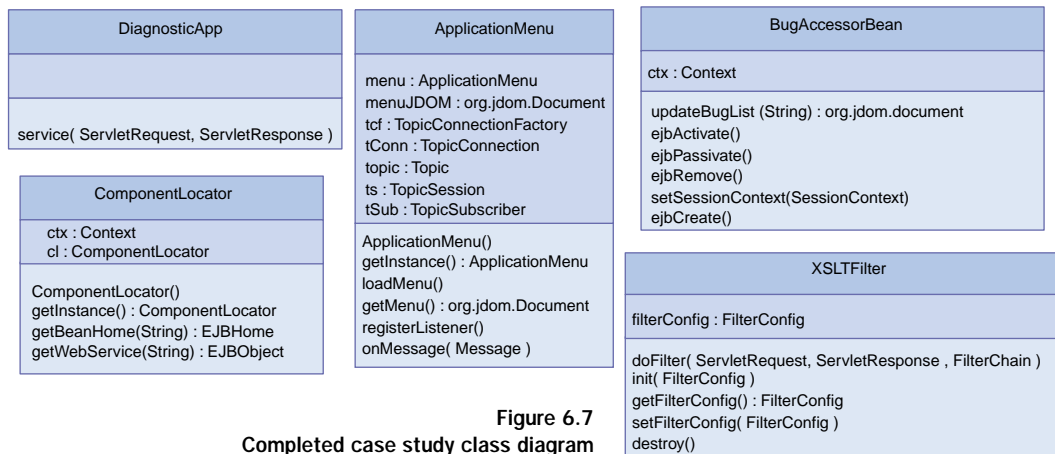


Figure 6.7
Completed case study class diagram

6.5 Validating our design

Before we move on, let's review the steps that we have taken so far. We began by choosing a suitable environment for our application. Next, we developed an analysis architecture that satisfied our case study requirements. Then, we added detail to those components during and added additional design level components necessary to make our application complete. Table 6.3 lists each component in our system and its purpose.

Table 6.3 Case Study Component Summary

Component	Purpose
BugAccessorBean	Communicates with partner web service.
ComponentLocator	Locates components and services.
ApplicationMenu	Caches application menu data.
DiagnosticApp	Controller servlet for application.
XSLTFilter	Performs XSLT transformation and rendering

There is one step left in the design process after the application of design patterns and the creation of our class diagrams. This step involves validating our architecture against the use cases. This ensures that the designed application satisfies the requirements. To perform our validation, we employ the use of a

sequence diagram. *Sequence diagrams* are a type of UML interaction diagram that show objects within the system interacting during the processing of a request. Figure 6.8 is a sequence diagram that shows the application flow for all three of our use cases.

The sequence diagram is read from top to bottom beginning with the first arrow, or message. Continuing down, you can see the flow of events between application components. Sequence diagrams may be documented in much more detail, including return values, but this level is sufficient for our purposes. As we move into the implementation section, you can trace each request along three paths—through the use case, the sequence diagram, and finally the code. Each object ① is an instance of the classes from our class

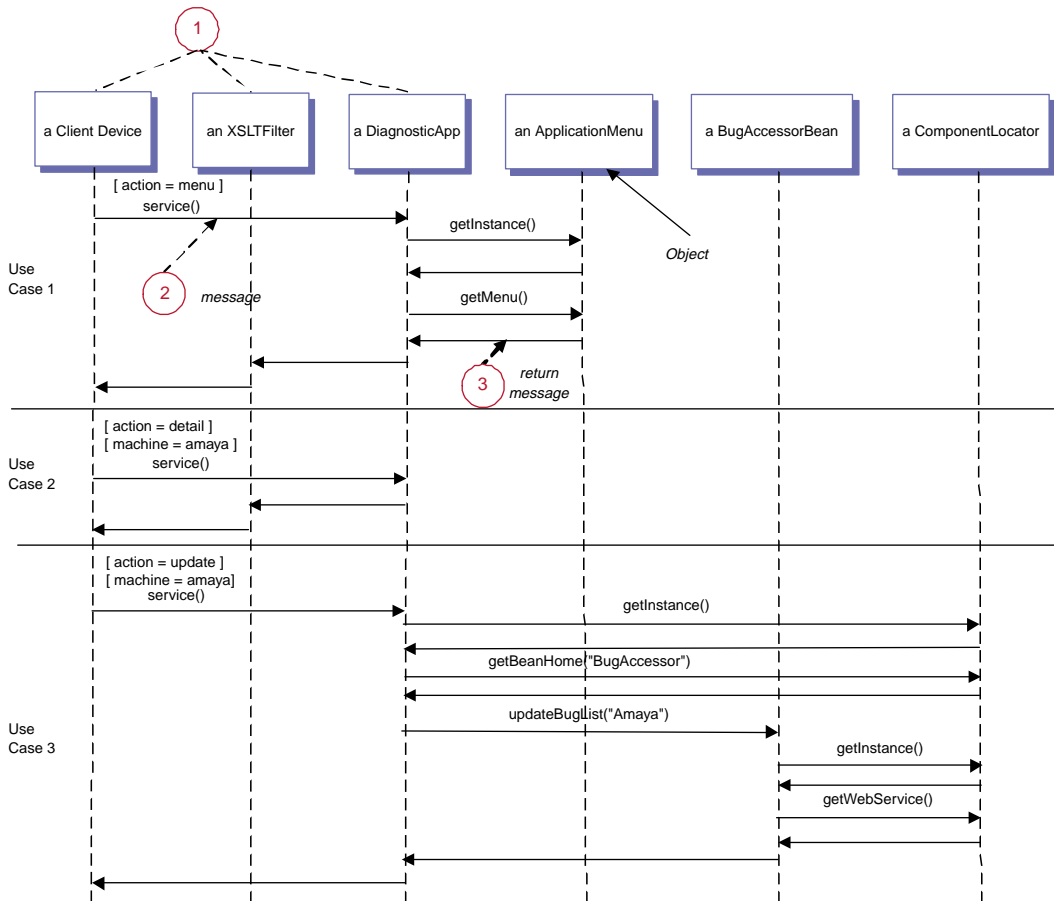


Figure 6.8 Sequence diagram for design validation

diagram. Right-pointing arrows ② are messages that indicate a method call. The method name is listed above each message. Left-pointing arrows ③ are return messages.

6.6 The implementation phase

With our design completed, we move into the implementation phase. In this section, we list and explain the interesting pieces of code to be found in the case study. As with the other examples in this book, we wrote this case study to illustrate certain concepts. If this were a true production application, some portions of the code would be more robust. For the full source code to this application, go to <http://www.manning.com/gabrick/>.

6.6.1 Building the controller servlet

The `DiagnosticApp` component is the controller servlet for our application. It accepts a parameter named `action` that is submitted on the query string of the client browser (`http://wwwcom/DiagnosticApp?action=menu`). The three possible values for this parameter are listed in table 6.4.

Table 6.4 Possible parameter values for `action` variable

Value	Purpose
menu	View application menu
detail	View detailed information about a specific machine
update	Contact manufacturer and update machine support information

Additionally, the machine name is passed to the servlet as a parameter if the action is `detail` or `update`. The `DiagnosticApp` servlet makes calls to the other business logic components to process the user's request. Once that processing is complete, the results are put into the request object along with the a reference to the appropriate stylesheet. The `XSLTFilter` (described in section 6.6.5) performs the XSLT processing and renders the output to the user.

Listing 6.1 DiagnosticApp servlet

```
public class DiagnosticApp extends HttpServlet {
    ...
    public void service( ServletRequest req, ServletResponse res )
        throws ServletException, IOException {
```

```

try {
    // If the user requested action is menu, then get the menu.
    String action = req.getParameter("action");

    if ( action.equals("menu") ) {
        ApplicationMenu menu
            = ApplicationMenu.getInstance();

        Document menuDoc = menu.getMenu();

        req.setAttribute( "outputDoc", menuDoc );
        req.setAttribute( "stylesheet",
            "config/mydomain/applications/book/menu.xsl" );
    }

    // If the user requested action is update, then have the BugAccessorBean
    // update the list.
    String machineName = req.getParameter( "machine" );
    if ( action.equals("update") ) {
        ComponentLocator cl = ComponentLocator.getInstance();
        BugAccessorHome bHome =
            (BugAccessorHome)
                cl.getBeanHome("examples.chapter6.BugAccessor");
        BugAccessor bugBean = bHome.create();
        Document detailDoc
            = bugBean.updateBugList( machineName );
        req.setAttribute( "outputDoc", detailDoc );
        req.setAttribute( "stylesheet",
            "config/mydomain/applications/book/bugs.xsl" );
    }

    // If the user requested action is detail, then get the list detail.
    // for the specified machine.
    if ( action.equals("detail") ) {
        SAXBuilder sBuilder = new SAXBuilder();
        Document detailDoc = sBuilder.build(new File(
            "config/mydomain/applications/book/" + machineName +
            ".xml"));
        req.setAttribute( "outputDoc", detailDoc );
        req.setAttribute( "stylesheet",
            "config/mydomain/applications/book/bugs.xsl" );
    }
}
...
}
...
}

```

Central
point for
locating
services

1

2 Communicates
with web service

- ❶ The `ComponentLocator` is a central point for locating services. Here we use it to locate the `BugAccessorHome`.
- ❷ Use the `BugAccessorBean` to communicate with the web service and update the list of issues for a particular machine.

6.6.2 Building the `ApplicationMenu` component

The next listing that we examine is the `ApplicationMenu` class. It is responsible for caching the list of available options for our home page. The `DiagnosticApp` servlet calls the `getMenu` method each time the menu is required. We have omitted the code that makes this component a singleton from this listing. It is identical to the code that we use in the `ComponentLocator` class to implement the Singleton pattern. We discuss the `ComponentLocator` code in section 6.6.3.

The `ApplicationMenu` also requires an administrative interface. We use this interface to reload the menu data from `menu.xml`. The class registers itself with a JMS Queue in the `registerListener` method. If `menu.xml` is updated to include an additional function, we need only put a message on the JMS queue. As a subscriber, the `ApplicationMenu` class will reload the menu data.

Listing 6.2 `ApplicationMenu` code

```
public class ApplicationMenu implements MessageListener {
    ...

    private void loadMenu() {
        try {
            SAXBuilder sBuilder = new SAXBuilder();
            menuJDOM = sBuilder.build(new
                File("config/mydomain/applications/book/menu.xml"));
        } catch (Exception ex) {
            System.out.println("Error in loadMenu" + ex.getMessage());
        }
    }

    public Document getMenu() {
        return menuJDOM;
    }

    private void registerListener() {
        try {
            Context ctx = new InitialContext();
            tcf = (TopicConnectionFactory)
                ctx.lookup("TopicConnectionFactory");
            topic = (Topic) ctx.lookup("MenuRefresh");
            tConn = tcf.createTopicConnection();
        }
    }
}
```

❶ Loads the menu from the file system

❷ Registers with the JMS Queue


```

        ts = tConn.createTopicSession( false,
            Session.AUTO_ACKNOWLEDGE );
        tSub = ts.createSubscriber( topic );
        tSub.setMessageListener( this );
        tConn.start();
    } catch (Exception ex) {
        System.out.println("Error in registerListener"
            + ex.getMessage());
    }
}

/**
 * Reload the menu data when a message is
 * received from the JMS Queue.
 */
public void onMessage (Message ms) {
    loadMenu();
}
...
}

```

↑
②

③ Message in
JMS Queue
triggers this
method

- ① This method loads the menu initially and is called whenever the menu needs to be reloaded.
- ② This code subscribes the `ApplicationMenu` to the `JMS Queue` so that the menu can be reloaded if the administrator updates it.
- ③ When the message is received from the `JMS Queue`, reload the menu.

6.6.3 Building the `ComponentLocator`

The `ComponentLocator` is responsible for locating remote objects and services throughout our application. It hides the complexity of the JNDI calls from the other classes and avoids the duplication of the code necessary to locate remote objects.

The private constructor and public `getInstance` methods are the way that this class implements the Singleton pattern. This class contains a static variable of type `ComponentLocator` that is initially set to null. On the first call to `getInstance`, the private constructor is called and the naming context is cached so that we do not need to reload it on subsequent calls. The static variable is also set to the instance of this class that we just created and all subsequent calls to `getInstance` merely return our singleton instance. See listing 6.3.

Listing 6.3 ComponentLocator implementation

```
public class ComponentLocator {

    private static ComponentLocator cl = null;
    private Context ctx = null;

    /**
     * Private constructor that is called only when there
     * is no other instance of this class.
     * Initializes and caches the context.
     */
    private ComponentLocator() {
        try {
            Properties prop = new Properties();
            prop.load(
                getClass().getResourceAsStream( "/jndi.properties "));
            ctx = new InitialContext( prop );
        } catch (Exception e) {
            System.out.println("Exception creating ComponentLocator;");
        }
    }

    /**
     * Method called to retrieve access to
     * the singleton instance. One will be
     * created if it does not already exist.
     */
    public static ComponentLocator getInstance() {
        if (cl == null ) {
            cl = new ComponentLocator();
        }
        return cl; |
    }

    /**
     * Method called by clients looking for EJBs
     */
    public EJBHome getBeanHome( String beanName ) {
        EJBHome ejbh = null;
        try {
            Object o = ctx.lookup( beanName );
            if ( o != null) {
                Class beanHomeClass
                    = Class.forName( beanName + Home );
                ejbh = (EJBHome)
                    PortableRemoteObject.narrow( o, beanHomeClass );
            }
        } catch (Exception e) {
            System.out.println(
                "Error in getBeanHome in ComponentLocator class"
                + e.getMessage());
        }
    }
}
```

① Caches the naming context

② Creates an instance if one has not already been created

③ Performs JNDI lookup to find bean home

```

    }
    return ejbh;
}

/**
 * Method called to locate web service for a particular
 * manufacturer. In this case, Only one is implemented.
 */
public EJBObject getWebService( String machineName ) {
    try {
        String className =
            CommonIssues.class.getName(); 4 Amaya's web service bean

        Properties prop = new Properties();
        prop.put( Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.soap.http.SoaInitialContextFactory" );
        prop.put( "weblogic.soap.wsdl.interface", className );
        prop.put( "weblogic.soap.verbose", "true" );
        // Register encoding types
        CodecFactory cf = CodecFactory.newInstance();
        cf.register( "http://schemas.xmlsoap.org/soap/encoding/",
            new SoapEncodingCodec() );
        cf.register( "http://xml.apache.org/xml-soap/literalxml",
            new LiteralCodec() );
        cf.register( "http://schemas.xmlsoap.org/soap/encoding/",
            new LiteralCodec() );
        cf.register( "http://xml.apache.org/xml-soap/literalxml",
            new SoapEncodingCodec() );
        prop.put( "weblogic.soap.encoding.factory", cf );
        Context ctx = new InitialContext( prop );
        CommonIssues webService = (CommonIssues)ctx.lookup
            ("http://localhost/issues/CommonIssues/CommonIssues.wsdl"); 5
        return webService;
    } catch (SoapFault fault) { Looks up web service
        System.out.println("Soap fault generated: " + fault);
        fault.printStackTrace();
    } catch (Exception e) {
        System.out.println("Error in getWebService." + e.getMessage());
    }
    return null;
}
}

```

- ❶ This code is called only once to cache the naming context in the instance of this class. It reads JNDI connection properties from a file named `jndi.properties`, which is found on the classpath.
- ❷ Tests the static variable that indicates whether an instance has already been created. If it has, then returns it. If not, creates an instance and then returns it to the caller.

- ③ This method is responsible for remote lookups of our bean homes. It locates a named object using the specified parameter and, if such an object is found, returns a reference to it.
- ④ The `CommonIssuesBean` is responsible for receiving web service calls at Amaya and returning the latest information.

Important details about the ComponentLocator

The `getWebService` method in the `ComponentLocator` class is the first piece of code that explicitly references `WebLogic` classes. In chapter 4, section 4.3.1, we discussed the architecture of an RPC-style web service. It involves using a stateless session bean to call a servlet that handles the SOAP communication with the web service. In our implementation, `WebLogic` has provided an implementation of that servlet in the `webllogic.soap.servlet` package. It shields the developer from much of the detail requiring the simple calling code that is listed in our method. If you were to port this code to another application server or choose to use another SOAP server, this code would have to be modified.

There are two other lines in the `getWebService` method that require further explanation. The first line in the method refers to a class called `CommonIssues`. There are two methods for invoking an RPC-style web service. You may use a static client to invoke a web service if you have the interfaces to the EJB, parameters, and return types associated with the web service. If not, you may write a dynamic client, which does not explicitly reference those classes in your code. In our case, Amaya implemented their web service using a stateless session bean called `CommonIssuesBean`. They have provided us with the interface and as a result, we have written our `getWebService` as a static client. Our reference to the `CommonIssues` class is a reference to the remote interface of their bean.

Finally, we provide a URL for Amaya's web service. The URL begins with `http://localhost`. This enables you to run the case study on one machine. As we will see when we deploy the application, the Amaya web service is running as a separate application. If you choose to deploy that portion of the application to another machine, merely change the URL in the `ComponentLocator` class to point to the appropriate location for the WSDL.

6.6.4 Building the BugAccessorBean

The `BugAccessorBean` is responsible for accessing our partner web services and updating the list of common problems associated with each computer we repair. It uses the `ComponentLocator` to locate the web service and then requests the

latest set of problems in the form of an `org.w3c.dom.Document`. This information is then written to the file system and returned to the calling component. In this listing, we include only the method that communicates with the web service and performs the update. The rest of the source code for this bean including the home and remote interfaces is very standard. See listing 6.4.

Listing 6.4 BugAccessorBean implementation

```
public class BugAccessorBean implements SessionBean {
    ...
    /*
     * Based on the name of the machine that is passed in,
     * call the web service and update the bug list.
     */
    public Document updateBugList( String name ) throws RemoteException {
        try {
            SAXBuilder sBuilder = new SAXBuilder();
            String fileName = name + ".xml";
            Document bugDoc = sBuilder.build(new
                File("config/mydomain/applications/book/"
                    + fileName));
            Element root = bugDoc.getRootElement();
            String lastUpdatedString
                = root.getChildText("last_updated");

            SimpleDateFormat formatter
                = new SimpleDateFormat("MM-dd-yy");
            ParsePosition pos = new ParsePosition(0);
            Date lastUpdated =
                formatter.parse( lastUpdatedString, pos );
            Date currentDate = new Date();
            String today = formatter.format( currentDate );
            Element dateElement = root.getChild("last_updated");
            dateElement.setText( today );

            ComponentLocator cl
                = ComponentLocator.getInstance();
            CommonIssues webService
                = (CommonIssues) cl.getWebService( name );
            org.w3c.dom.Document wsDoc
                = webService.getIssuesList("amaya");

            DOMBuilder dBuilder = new DOMBuilder();
            Document newIssueDoc = dBuilder.build( wsDoc );
            Element docRoot = newIssueDoc.getRootElement();
            Element newIssues = docRoot.getChild( "issues" );
            newIssues.detach();

            Element issues = root.getChild("issues");
            root.removeContent( issues );
```

1 Retrieves current list of detailed issues

Updates the last_updated element to today's date

2 Communicates with the web service

← Replaces the old set of issues with the updated set of issues

```

root.addContent( newIssues );

XMLOutputter xmlOut = new XMLOutputter();
xmlOut.output( bugDoc,
    new FileOutputStream( new
        File("config/mydomain/applications/book/"
            + fileName) ) );
return bugDoc;
} catch (Exception e) {
    System.out.println("Error in update bug list bean."
        + e.getMessage());
}
return null;
}
}

```

- ❶ A detailed list of common problems associated with each machine is maintained in XML files on the file system. This code retrieves this list for a given machine.
- ❷ This code communicates with Amaya's web service. A string parameter is passed to the `getIssuesList` method that represents the name of the computer that we're requesting information about. This method returns an `org.w3c.dom.Document` containing the list of problems associated with that computer.

6.6.5 Building the XSLTFilter

Finally, we list the code for the `XSLTFilter` class. The `XSLTFilter` is responsible for transforming the output of our business logic to be rendered to the client device. This implementation is a simplified version of the `XSLTFilter` listed in chapter 4 as we are currently only concerned with one output format.

Listing 6.5 XSLTFilter implementation

```

public class XSLTFilter implements Filter {
    ...
    /*
     * Method used to transform response
     * prior to rendering.
     */
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        try {
            chain.doFilter(request, response); ❶ Sends processing
                                                to servlet
            Document outputDoc = (Document)
                request.getAttribute( "outputDoc" ); ❷ Performs XSLT transforma-
                                                        tion on results of servlet
        }
    }
}

```

```

String stylesheet = (String)
    request.getAttribute("stylesheet");
Transformer trans = TransformerFactory.newInstance()
    .newTransformer( new StreamSource(stylesheet) );
JDOMResult result = new JDOMResult();
trans.transform(new JDOMSource(outputDoc), result);
Document resultDoc = result.getDocument();

response.setContentType("text/vnd.wap.wml");
PrintWriter out = response.getWriter();
XMLOutputter xOut = new XMLOutputter();
xOut.output( resultDoc, out );
} catch (Exception e) {
    System.out.println("Error in XSLTFilter: " + e.getMessage());
}
}
}

```

↑
2

← Renders the transformed output to the user

- ❶ This line sends the processing of this request to the next filter in the filter chain. In this case, we only have one filter so the processing is passed along to the `DiagnosticApp` servlet.
- ❷ Once the processing has returned from the `DiagnosticApp` servlet, the filter performs the XSLT transformation using the stylesheet and XML document in the request object.

6.7 Structuring application data

After reviewing the code for our application, the next step is to examine the data. We have intentionally left the structure of the XML simple so that you can follow the flow of data through the application. Listing 6.6 is the `menu.xml` file containing menu data for the application.

Listing 6.6 The `menu.xml` data file

```

<?xml version="1.0" encoding="UTF-8"?>
<menu>
  <option>Diagnose Hardware Problem</option>
  <option>View Availability of Replacement System</option>
  <option>Order Replacement Parts</option>
  <option>View Common System Problems</option>
</machines>

```

List of
initial
menu
options

```
<machine id="ominco4000">Omnicco 4000</machine>
<machine id="amaya">Amaya Workstation</machine>
<machine id="computech">Computech ATX</machine>
<machine id="xpad">ACME xPad</machine>
</machines>
</menu>
```

List of machines
that RepairCo
supports

Listing 6.7 is `amaya.xml`, which contains a detailed list of problems with the Amaya machine. Given the scope of our application, it is not necessary to validate these files against DTDs or XML Schemas. This would certainly be necessary in a production application. Additionally, a backup of the `amaya.xml` file should be made each time the list is updated using Amaya's web service. In the event that the update fails, you can revert to the previous version.

Listing 6.7 The `amaya.xml` data file

```
<?xml version="1.0" encoding="UTF-8"?>
<amaya>
  <machine>amaya</machine>
  <last_updated>06-01-01</last_updated>
  <issues>
    <issue id="0001">Machine Is Not Receiving Power</issue>
    <issue id="0002">LCD Stopped Working</issue>
    <issue id="0003">Abnormally Low Battery Life</issue>
  </issues>
</amaya>
```

List of
common
problems
with
Amaya
machine

6.8 The Amaya web service

Though we have completed our analysis and implementation of the RepairCo side of the case study, we are fortunate enough to have access to Amaya's web service implementation as well. This web service provides Amaya resellers with access to the latest support information about its machines. This particular web service contains one stateless session bean (`CommonIssuesBean`) running in a J2EE container that returns the latest data from an XML document (`amaya.xml`) on the file system. This scenario is depicted in figure 6.9.

The client in this scenario ① can be anything capable of sending a SOAP request. In this scenario, the client is our repair diagnostic application. It sends a single string parameter with the SOAP request, which represents the name of the machine for which we are requesting information. The SOAP server at

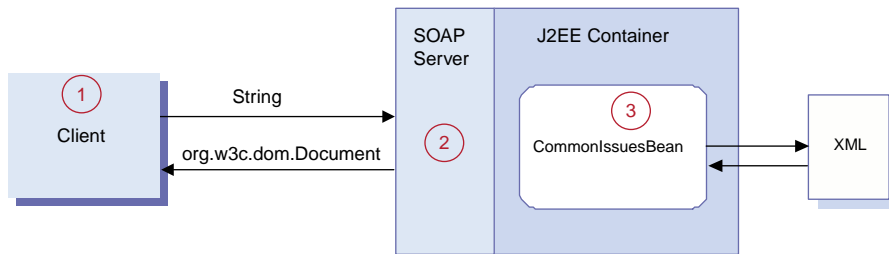


Figure 6.9 Amaya web service integration

Amaya ② receives SOAP requests and passes them to the appropriate component. In this case, the `CommonIssuesBean` is the target of our SOAP request. The `CommonIssuesBean` ③ accesses XML data regarding support information for Amaya's computers. It returns an `org.w3c.dom.Document` to the SOAP server, which in turn sends this information back to the requesting client.

Listing 6.8 is the implementation of the `CommonIssuesBean`. As with our previous listing, we omit the source for the home and remote interfaces.

Listing 6.8 `CommonIssuesBean` implementation

```

public class CommonIssuesBean implements SessionBean {
    ...
    /**
     * Get the list of issues from the file system
     * and return the XML document.
     */
    public Document getIssuesList( String name ) throws RemoteException {
        try {
            DocumentBuilderFactory dFactory
                = DocumentBuilderFactory.newInstance();
            DocumentBuilder dBuilder
                = dFactory.newDocumentBuilder();
            Document issueDoc = dBuilder.parse
                ("config/mydomain/applications/amaya/"
                 + name + ".xml");
            return issueDoc; |#1
        } catch (Exception e) {
            System.out.println("Error in common issues bean: " + e.getMessage());
        }
        return null;
    }
}
  
```

Retrieves document from file system and returns it

The data that this bean sends back to RepairCo (`amaya.xml`) is listing 6.9. Notice that it contains an issue with the `id` equal to 0004; an additional problem with the Amaya Workstation that RepairCo's list does not contain (see listing 6.7). This issue will be added to the list when RepairCo calls the Amaya web service.

Listing 6.9 Amaya's issues list with the Amaya machine

```
<?xml version="1.0" encoding="UTF-8"?>
<amaya>
  <issues>
    <issue id="0001">Machine Is Not Receiving Power</issue>
    <issue id="0002">LCD Stopped Working</issue>
    <issue id="0003">Abnormally Low Battery Life</issue>
    <issue id="0004">LCD Flickers</issue>
  </issues>
</amaya>
```

The final portion of the Amaya implementation that we examine is the WSDL for the web service. This should look familiar due to its similarity to the WSDL that we examined in chapter 4. WebLogic generated the file in listing 6.10 automatically, alleviating the need for us to create it manually.

Listing 6.10 WSDL for Amaya web service

```
<?xml version="1.0"?>
<definitions
  targetNamespace="java:amaya.webservices"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:dom="http://www.w3c.org/1999/DOM"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:tns="java:amaya.webservices"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

  <types>
    <schema targetNamespace='java:amaya.webservices'
      xmlns='http://www.w3.org/1999/XMLSchema' >
    </schema>
  </types>

  <message name="getIssuesListRequest">
    <part name="arg0" type="xsd:string" />
  </message>
  <message name="getIssuesListResponse">
    <part name="return" type="dom:org.w3c.dom.Document" />
  </message>
```

Derived XML
data types
would go here

Inbound/Outbound
message definitions
for our RPC service

```

<portType name="CommonIssuesPortType">
  <operation name="getIssuesList">
    <input message="tns:getIssuesListRequest"/>
    <output message="tns:getIssuesListResponse"/>
  </operation>
</portType>

<binding
  name="CommonIssuesBinding"
  type="tns:CommonIssuesPortType">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getIssuesList">
    <soap:operation soapAction="urn:getIssuesList"/>
    <input>
      <soap:body use="encoded" namespace='urn:CommonIssues'
        encodingStyle=
          "http://xml.apache.org/xml-soap/literalxml" ⚡
          http://schemas.xmlsoap.org/soap/encoding/" />
    </input>
    <output>
      <soap:body use="encoded" namespace='urn:CommonIssues'
        encodingStyle=
          "http://xml.apache.org/xml-soap/literalxml" ⚡
          http://schemas.xmlsoap.org/soap/encoding/" />
    </output>
  </operation>
</binding>

<service
  name="CommonIssues">
  <documentation>
    This service provides manufacturer support information
    on our products.
  </documentation>
  <port name="CommonIssuesPort"
    binding="tns:CommonIssuesBinding">
    <soap:address
      location="http://localhost:80/issues/issues"/>
  </port>
</service>
</definitions>

```

❶ Port type definition

A binding of our new port type to SOAP/HTTP

Web service definition ❷

- ❶ The definition of the web service port type. This binds the messages together into an RPC request/response operation.
- ❷ The actual web service definition, built from the message, port, and binding information in this file. This is where the actual URL for the service is specified.

6.9 *Running the application*

Now that we have designed and implemented our application, we can run it and examine its output. As we stated earlier, the implementation is intended to run on WebLogic version 6.1. For those of you who wish to port it to another platform, we have included all of the source code for our application online. BEA has a comprehensive set of documentation on their web site, so we do not include product installation or administration information. Please refer to <http://edocs.bea.com> for WebLogic documentation.

6.9.1 *Installation*

As for the installation of our application, first go to <http://www.manning.com/gabrick/> and download the Zip file containing the source as well as the compiled code for the application. Unzip the file under the directory listed in figure 6.10. The resulting directory structure is shown in figure 6.10.

The directory structure in figure 6.10 is the default directory structure for a WebLogic 6.1 installation. Once you have unpacked the files, you may start WebLogic. From the administration console, which is located by default at <http://localhost:7001/console>, you must perform the following actions:

- 1 Under the Deployments > Applications folder add an application:
Path - `.\config\mydomain\applications\amaya\issuesWebService.ear`
- 2 Under the Deployments > EJB folder, configure an EJB with the following fields:
URI: `bug_bean_ejb.jar`
Path: `.\config\mydomain\applications\book\WEB-INF\lib`
- 3 Under the Deployments > Web Applications folder, add a web application with the following fields:
Name: `book`
URI: `book`
Path: `.\config\mydomain\applications`

Once you have completed these actions, the application is deployed and we can view the output of our case study. If you have trouble deploying the application components, refer to the Administration Guide within the WebLogic documentation.

There is one portion of our environment that we have not yet configured. The client side that we discussed in section 6.2 consists of the Palm emulator

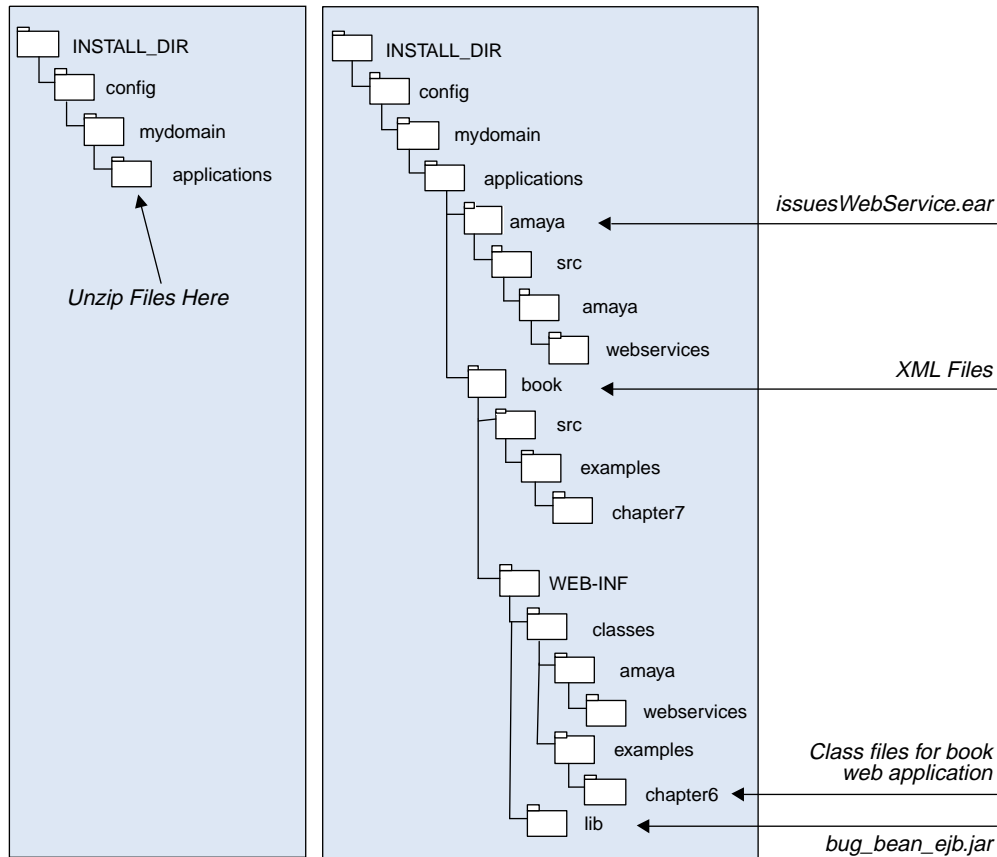


Figure 6.10 Directory structure for case study

with the EzWAP browser that renders WML. If you are not able to re-create this environment, all you need is a browser that renders WML. In lieu of that, you may also use a web browser such as Internet Explorer and examine the contents of the response.

6.9.2 Viewing the main menu

To load the initial screen of our application, enter the URL `http://localhost:7001/book/DiagnosticApp?action=menu`. This request calls the `DiagnosticApp` servlet, which in turn loads the `ApplicationMenu` and returns it to the user. Figure 6.11 depicts the results of this request. The right side of this picture is the list of menu options.

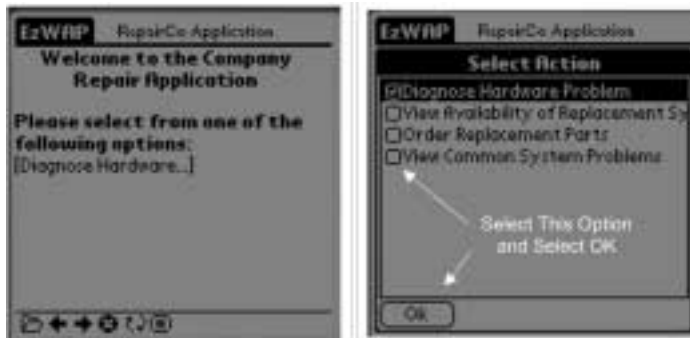


Figure 6.11 Case study results: Viewing the Application menu

6.9.3 Viewing common system problems

Once you select View Common System Problems, a list of machines that RepairCo services is loaded. This is depicted in figure 6.12.



Figure 6.12 Case study results: List of machines that RepairCo services

6.9.4 Viewing and updating the Amaya problem list

The next request can be initiated by selecting Amaya Workstation or by entering the URL <http://localhost:7001/book/DiagnosticApp?action=detail&machine=amaya>. This loads RepairCo's information regarding problems with the Amaya machine. The results are depicted in figure 6.13 on the left side. Technicians who don't find the needed information in this list may click on the Update link. This calls Amaya's web service, which updates the list on the right side of figure 6.13. Notice the additional problem, LCD Flickers,

that was returned to our application in the updated list. You may call the web service directly with the URL `http://localhost:7001/book/DiagnosticApp?action=update&machine=amaya`.

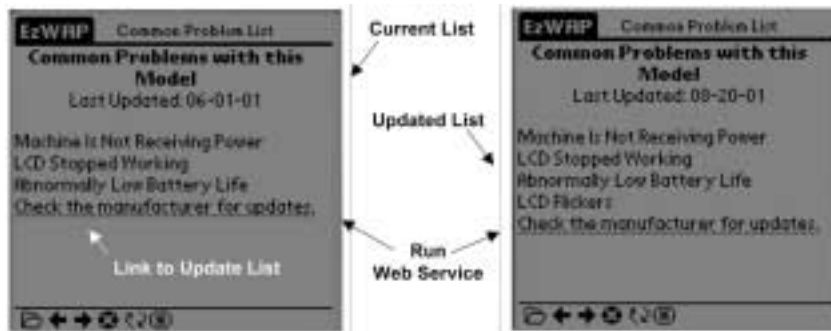


Figure 6.13 Case study results: Common list of problems with Amaya machine

6.9.5 Inspecting the web services SOAP messages

Listing 6.11 contains the SOAP messages that are sent between the RepairCo application and the Amaya application. They are very similar to the SOAP messages we reviewed in chapter 4, section 4.2.

Listing 6.11 SOAP messaging for Amaya web service

```
----- SENDING XML -----
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
  'http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:SOAP-ENC='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/1999/XMLSchema'>
  <SOAP-ENV:Body>
    <ns0:getIssuesList xmlns:ns0='urn:CommonIssues'
      SOAP-ENV:encodingStyle='http://xml.apache.org/xml-soap/literalxml
      http://schemas.xmlsoap.org/soap/encoding/'>
      <ns0:arg0>amaya</ns0:arg0>
    </ns0:getIssuesList>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

← Single parameter passed to the web service

```
----- RECEIVING XML -----
<?xml version='1.0' encoding='UTF-8'?>

<SOAP-ENV:Envelope xmlns:SOAP-ENV=
  'http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:SOAP-ENC='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/1999/XMLSchema'>

  <SOAP-ENV:Body>
    <ns0:getIssuesListResponse xmlns:ns0='urn:local'
      SOAP-ENV:encodingStyle='http://xml.apache.org/xml-soap/literalxml
      http://schemas.xmlsoap.org/soap/encoding/'>

      <ns0:return>

        <amaya>
          <issues>
            <issue id="0001">Machine Is Not Receiving Power</issue>
            <issue id="0002">LCD Stopped Working</issue>
            <issue id="0003">Abnormally Low Battery Life</issue>
            <issue id="0004">LCD Flickers</issue>
          </issues>
        </amaya>
      </ns0:return>
    </ns0:getIssuesListResponse>
  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

6.10 Summary

The purpose of this chapter was to demonstrate the concepts and technologies examined in this book through the use of a case study. We began by introducing RepairCo, a fictional company that services computers. They needed an application to enable their technicians to access data while they were in the field. We then proceeded to analyze and design these requirements, applying J2EE and XML patterns and concepts to our architecture. Our implementation made use of several J2EE design patterns including the Singleton pattern, Service Locator pattern, and the Decorating Filter pattern. Additionally, we used XML for our data storage and JDOM for manipulation within Java. Finally, we implemented an RPC-style web service and used it to integrate our application across companies. The result of our case study is a flexible, robust architecture with several components that can be easily adapted and reused in other applications.



Design patterns for J2EE and XML

This appendix

- Identifies J2EE design patterns for use with XML
- Groups patterns by application tier

This appendix contains a brief overview of a few J2EE and XML design patterns. If you are not familiar with design patterns generally or with J2EE design patterns, more detailed information is available on the Web and in the sources listed in the bibliography. A good place to start on the Web is the J2EE Blueprints design pattern section,

http://java.sun.com/j2ee/blueprints/design_patterns/.

Briefly stated, a software design pattern is a reusable blueprint for structuring code to solve a particular class of problem. Numerous design patterns specific to J2EE have been published, and there are even a couple of books on the subject. In this section, we introduce only those patterns that are used in this book. We describe each pattern and note where and how it is used in the chapters, including the role XML plays in each.

A.1 *Presentation layer patterns*

Patterns listed in this section can be applied to enhance the flexibility and extensibility of your application's web interface. They involve the J2EE web container, servlets, JSPs, and related objects.

A.1.1 *The Decorating Filter pattern*

The Decorating Filter pattern involves applying inbound and outbound filters to modify client request and response data across distinct request types. The pattern is depicted in figure A.1 and is fully supported by the servlet API as of version 2.3. The Decorating Filter is useful when a general piece of logic needs to be applied globally to many types of user interactions.

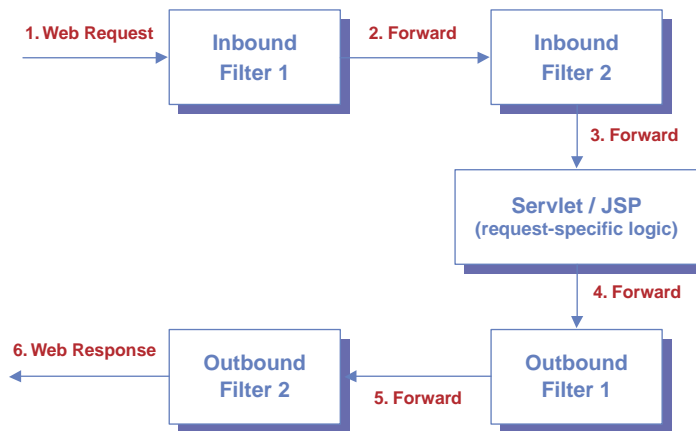


Figure A.1 The Decorating Filter pattern

For example, your application may need to determine whether a session object exists for a given user and create one if necessary. If the check needs to occur on every request, encapsulating this logic in an inbound filter makes sense. Perhaps your user interface components produce XML that needs to be translated into an output document based on the user's device type and locale. This logic could be encapsulated in an outbound filter and applied to every response. In chapter 5, we do this very thing using XSLT, a document transformation technology based on XML. This same pattern is used with XSLT the chapter 6 case study.

There are many other circumstances where this pattern can be useful. In general, you will use inbound filters to prepare requests for processing and outbound filters for output formatting. Other functionality, such as logging, could be implemented as either type of filter, depending on the data you wish to collect.

A.1.2 *The Model-View-Controller pattern*

This pattern is familiar to many developers who have written applications requiring multiple user views of the same data. The concept here is that access to an application is centrally managed by a controller component, which often performs security checks and logs request activity. All client requests pass through the controller and are dispatched to a component that renders the user interface. This component is known as the view, and provides a specific user with customized access to the application and its data. Internally, the application actually has a single state (the model) from which all views are derived. This is depicted in figure A.2.

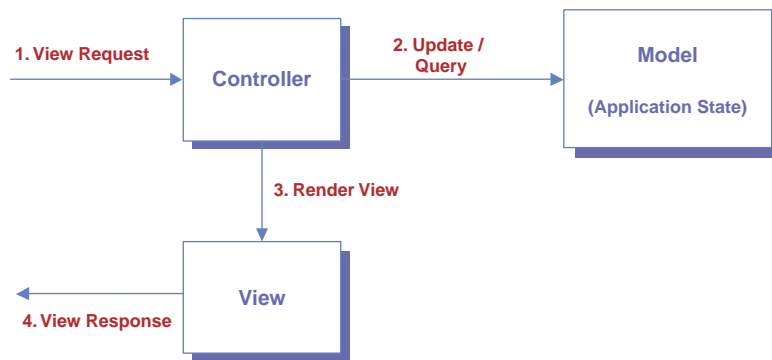


Figure A.2 The Model-View-Controller pattern

Many J2EE presentation layers employ the Model-View-Controller (MVC) pattern because of the common distributed application requirement of serving many types of users with different access rights and functional needs from a single user interface. As such, the MVC pattern has been mapped onto the J2EE web components in various ways, which we will characterize generally here.

A servlet most often acts as the controller component for the application, although a JSP page can be used in some cases. This controller (often referred to as a *front controller*) performs general functionality, such as access control and request logging. It may then employ one or more helper components to assist in request-specific processing. These can be virtually any type of Java class, but are often JDBC- or EJB-aware components that integrate with other tiers of the application.

Once request processing has been completed, the request and response data are forwarded to the view component. This is usually a JSP. The view component in turn uses helpers to assist in rendering the response data in an appropriate format. These helper classes are usually custom tags and JavaBeans. A full implementation of the MVC pattern in J2EE is depicted in figure A.3. The examples in chapter 5 use this pattern, but substitute XML technologies for certain J2EE components to make the solution even more robust. The case study in chapter 6 uses the MVC pattern with an XSLT servlet filter as the view component.

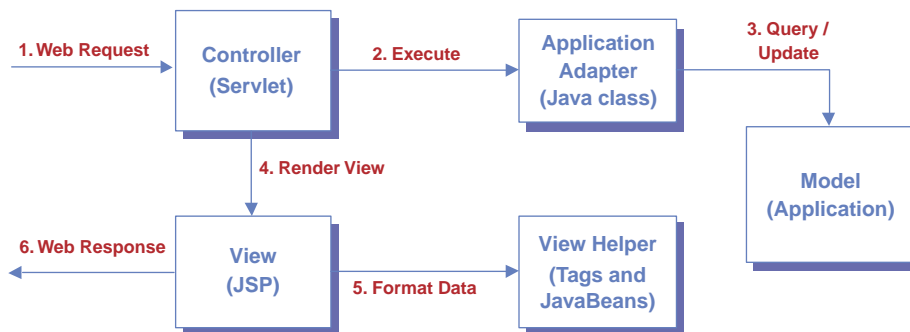


Figure A.3 J2EE Model-View-Controller architecture

There are numerous variations on the MVC theme in J2EE, and not every component shown in figure A.3 is required in every solution. Also keep in mind that it is easy to get carried away with the MVC pattern and design a solution that is overly complicated and difficult for developers to use and comprehend.

A.2 Application- and service-layer patterns

Patterns in this section are focused on increasing the flexibility, maintainability, and modularity of your internal application components. The focus at this layer is on hiding the complexities that lie below the client APIs, centralizing configuration management, and creating tightly cohered, loosely coupled application components.

A.2.1 The Service Locator pattern

The Service Locator pattern is one of the simplest and most useful of J2EE patterns. This pattern hides the complexity of locating remote services, such as EJBs, data sources, message queues, and mail servers. When employed properly, it can also reduce administrative burdens on the application assembler and deployer.

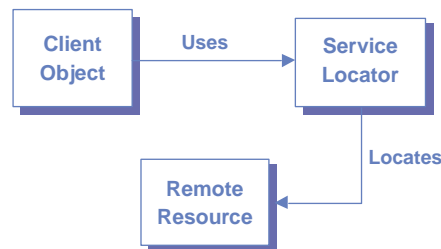


Figure A.4 The Service Locator pattern

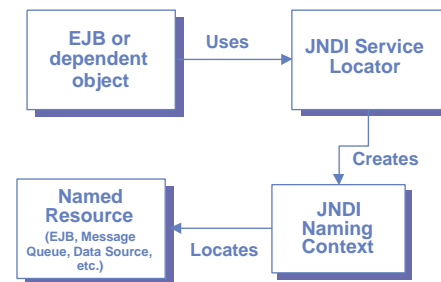


Figure A.5 A JNDI Service Locator implementation

The concept behind this pattern is to provide a simple API for obtaining a reference to a service component. This is depicted in figure A.4. For example, a JNDI service locator might provide a simple lookup method for an instance of a particular `EJBHome` class. While there are numerous steps involved in the JNDI lookup process, the Service Locator's clients are shielded from the underlying complexity. This is depicted in figure A.5.

The Service Locator can also manage mappings of identifiers used in the application to resources in the operational environment. For example, a Service Locator might contain a map between an object's class and a JNDI name in the deployed configuration. This technique is useful for centrally managing JNDI configuration information throughout your application, and is demonstrated in the case study in chapter 6.

A.2.2 The Business Delegate pattern

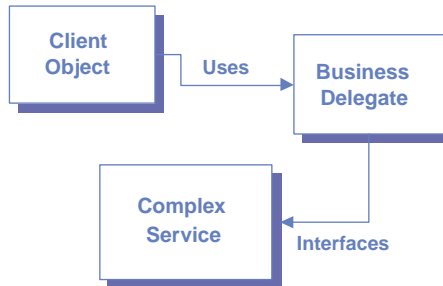


Figure A.6 The Business Delegate pattern

The Business Delegate pattern is employed to simplify client access to a particular application service. In this pattern, a Business Delegate object acts as a proxy between application clients and the service. This is depicted in figure A.6. This pattern is related to the more general Proxy and Adapter software patterns and hides the complexity of locating and accessing a complicated business API from the client.

Often, the Business Delegate utilizes a Service Locator of some kind to locate a particular service and connect to it.

In J2EE, Business Delegates are often dependent objects used by servlets and EJBs. For example, an adapter class might provide access to an internal ERP system for querying order status. This adapter might locate the ERP system using JNDI parameters and engage in a complicated interaction with the remote system. All this is transparent to the objects that use the adapter.

The Business Delegate pattern is demonstrated by the JAXM examples in chapter 4, which use it to shield application components from the complexities of interacting with remote applications via SOAP.

A.2.3 The Value Object pattern

The Value Object pattern is a simple but very useful pattern in J2EE. A value object is a serializable representation of a complex set of data suitable for passing between application tiers via RMI. Value objects are generally utility classes, similar to structs in C, but can contain behavior to validate or prohibit internal data modification as well. An example of this pattern is creating an `OrderInfo` object that contains all the data about a given order. This object could be serialized across the network and used to display information to the user. This is depicted in figure A.7.

Value objects are critical to the proper operation of an RMI-based component model such as EJB. Since local references to remote data are not possible, a snapshot of the data is passed by value to the remote client. Value objects are employed very often. See the case study for a detailed example of their use with XML structures.

There are two other J2EE patterns used with value objects. For completeness, we mention them here. The Value Object Assembler is a pattern for

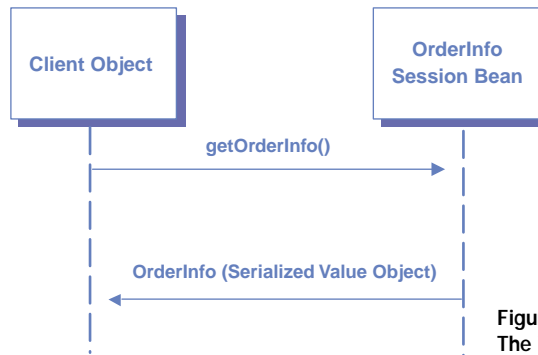


Figure A.7
The Value Object pattern

composing complicated value objects from various other objects. The Value List Handler is a pattern for creating, manipulating, and updating sets of value objects at the same time. The details of these patterns are available at Sun's J2EE patterns web site.

A.2.4 The Data Access Object pattern

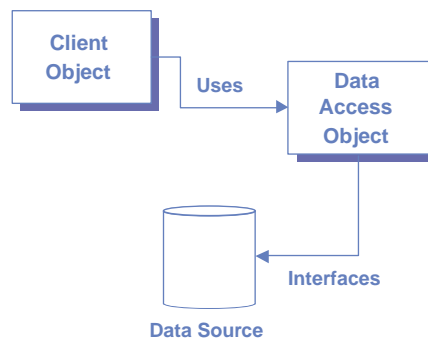


Figure A.8 The Data Access Object pattern

The Data Access Object pattern is useful for interacting synchronously with a persistent data store. This pattern is commonly used to decouple application components, such as EJBs and servlets, from an underlying database. The data access object handles the complexity of interacting with the data source and provides a simple interface for its application component clients to use. This pattern is depicted in figure A.8.

Often, a data access object is a JDBC-aware class that interacts with a specific relational database, but the pattern can also be applied to other types of enterprise systems. In the latter case, the data access object can be a wrapper for things such as ERP systems, legacy mainframe applications, and other proprietary applications.

You may be wondering why a data access object would be used to wrap external systems instead of using an entity EJB. The reason is that entity EJBs are not intended to simply act as proxies for calls to external systems. An entity

EJB is appropriate when your application must implement business rules and other logic in conjunction with access to shared, external data.

For example, if you are modeling a business workflow, an entity EJB might be appropriate. This is because many users require access to the workflow, the workflow itself must maintain its own internal state, and the workflow component must implement logic to ensure that transitions between states are validated.

Data access objects are appropriate when access is required on a per-client (or per-request) basis and most of the data manipulation and other logic is implemented by the remote system. For example, a remote procedure call to an ERP system to obtain a specific customer's product pricing is best handled by a data access object. This is because access to the customer pricing information does not need to be shared among many users and the logic to calculate the price is contained within the ERP system.

Knowing when to use a data access object instead of an entity bean will be invaluable to you, so take the time to understand this pattern and its implications thoroughly. This pattern is used in the examples in chapter 3 both for relational and XML-based data repositories.



Distributed application security

This appendix

- Summarizes important distributed security considerations
- Explains common security terms and technology

This appendix contains information about securing your J2EE and XML application from hackers and other unauthorized system users. It contains a detailed discussion of the security risks involved in distributed systems development and the use of cryptography and shared secrets to secure communication among system components.

B.1 *Securing distributed communication channels*

Passing messages between processes exposes the system to two types of risk. The first is that a third party will intercept a message containing sensitive data. The second is that a third party will falsify a message or alter a valid one before it is delivered to its intended recipient. Both of these risks present potentially serious threats, ranging from loss of system reliability to loss of your competitive advantage. To minimize these risks, communication between processes should be encrypted when appropriate. The cryptographic techniques described in section B.3 are used to do just that.

B.2 *Securing application components*

Even after communication has been secured, there are additional risks at the application level. One is that an external entity might access resources it should not. Another is that an external entity might gain access to resources by falsifying its identity. To address these concerns, every distributed application must have an authentication model and an authorization model.

B.2.1 *Authentication models*

Authentication is the process of ensuring that an entity is who it says it is. In its weakest form, an entity is required to provide a password to authenticate itself. Using something the entity knows to authenticate is not very secure, because any other entity that also knows this information can impersonate the real entity.

Stronger authentication mechanisms can be created using public key cryptography and certificate chains. In this model, a *certificate authority* that is trusted by your application vouches for the entity wishing to be authenticated, by signing the entity's authentication credential either directly or indirectly through a certificate chain. Your application agrees to transitively trust entities that are trusted by the certificate authority. This is a stronger form of authentication because it uses something the entity has (a signed credential), rather

than something the entity knows. Still, this model is susceptible to falsification of the credential, but this possibility is extremely remote.

The strongest form of authentication would be to use something the entity is. In the case of humans, biometric readings are an example. In a distributed environment, this level of security is costly to implement and relies at some level on the same credential mechanism described previously, since the remote entity is never physically present in a distributed environment.

B.2.2 Authorization models

Authorization is the process of granting some entity access to a given resource. It is closely related to authentication, because your system must believe that an entity is authentic before worrying about what that entity can and cannot do. However, it is also a distinct, second phase of remote resource access that allows an application to grant different levels of privileges to different users.

Authorization is commonly implemented using access control lists. The application maintains a list of entities or (more commonly) groups of entities that can access and/or modify a resource. These lists are often stored in flat files in small systems and in relational databases or directory servers in larger ones. These lists are a valuable resource themselves, which should be secured through encryption when possible. As these lists tend to grow exponentially as the numbers and types of entities in the system increase, using flat files becomes impractical quickly. Most systems of even modest size use dedicated software and hardware resources to manage this information. Examples include Lightweight Directory Access Protocol (LDAP) servers and network operating system databases.

The authorization scheme you choose and its implementation can have significant performance impacts on your system, if access to resources is fine-grained. If, for example, an authorization check needs to be done each time a particular request is serviced, this will significantly slow response times as usage levels increase. To combat this, consider retrieving and caching all of an entity's access rights in a convenient location when they first authenticate themselves.

B.2.3 Distributed security contexts

An important issue in security modeling is the requirement to share a common security context among system participants. If, for example, a requirement of your system is that a user authenticates only once but gains access to five independent applications within the system, those five applications need to share security data about users between them. If all five applications are administered by the same organization, storing access rights in a directory

server and forcing individual applications to defer to the directory server for user authentication and authorization might accomplish this. If, on the other hand, security data must be shared across organizations or public networks, the certificate chaining and cryptography techniques described in section B.3 may be more appropriate. For example, a user could authenticate to any individual application and have it vouch for their authenticity when communicating with other applications over secure channels.

B.3 *Using cryptography and shared secrets*

A critical dimension of distributed system security is ensuring the security and integrity of information exchange. To realize such exchanges, distributed systems generally employ cryptographic techniques and shared secrets. In a secure communication, both sender and receiver possess knowledge of a shared secret (or *key*) with which data to be transmitted is enciphered (by the sender) and/or deciphered (by the receiver). This key is usually a large number that is very hard to guess. In addition to encrypting the message, checksums and digital signatures based on the message data can be sent with the message to ensure at the receiving end that a message is both authentic and has not been modified during transmission.

B.3.1 *Symmetrical cryptography*

In symmetrical cryptography, both sender and receiver use the same, shared key to encipher and decipher data. This technique can be used effectively to exchange large amounts of secure data. However, the shared knowledge of the secret key in symmetrical cryptography threatens the security of the system. Since the key value needs to be known to multiple parties, the opportunities to have the key fall into the wrong hands increases with the size of the system. Also, the key itself can't be securely transmitted over the network to remote systems to begin a secure dialog.

B.3.2 *Asymmetrical cryptography*

To address these and other concerns, asymmetrical cryptography (also called *public key* cryptography) was developed. This method uses a pair of keys for each communication that are mathematically related to one another. The idea is that it is impossible to determine the value of one of the keys by examining the other. One of the keys is "private" to a particular entity, and is never shared with anyone. The other key is "public," and is freely available to anyone who wants it. Data encrypted with the *private key* can only be decrypted

with the *public key*, and vice versa. This can solve the problem of initiating secure dialogs and eliminate the security risk of sharing symmetrical keys among parties.

One key benefit of asymmetrical cryptography is the ability to create *digital signatures*. A digital signature is a *message digest*, similar to a checksum. It is based on the original contents of a message and is created using the message sender's private key. A digital signature, once verified, guarantees that a message originated from the specified sender and has not been altered during transmission. The use of digital signatures is especially important in banking applications, to facilitate *nonrepudiation* by financial account holders. (Nonrepudiation means that the originator of a transaction cannot later claim that someone else had used his or her account fraudulently and refuse to accept responsibility for the transaction.)

B.3.3 Tradeoffs and common implementations

While public key cryptography is more secure, it is also far slower than symmetrical cryptography. Therefore, it is common for security protocols to use a combination of the two, as in the case of the Secure Sockets Layer protocol (SSL). In such models, a secure communication session is established using public key cryptography. After this establishment, a symmetrical *session key* is agreed upon between the two parties communicating. This agreed-upon, temporary key is used to encrypt and decrypt messages for the remainder of the session. This combines the advantages of symmetrical and asymmetrical methods to make secure communication as fast and secure as possible.



The Ant build tool

This appendix

- Introduces the Apache Ant build tool
- Provides a starter build file
- Demonstrates development of custom tasks and build listeners

Ant is a build and configuration management tool written entirely in Java. It is similar in purpose and function to the UNIX make command-line utility, but is far easier to use and far more functional. The make utility, with which you may be familiar, is fussy about tab characters and spaces, very much platform-dependent, and can be cryptic to use at times.

Because Ant is Java-based, it is completely platform independent. It uses an XML configuration file that is easy to build and manage. Furthermore, Ant is completely open. Not only can you obtain the source code for the base tool, it is simple to extend the functionality of Ant by writing custom tasks to do specific things you require in your environment.

DEFINITION A *task* is a Java component that is invoked by the Ant build process. It is designed to be generic and reusable across development projects.

The need to write custom tasks is infrequent and becoming even less frequent. Ant has quickly gained popularity and has been extended already by many developers in a variety of ways. Table C.1 summarizes the tasks that are already built into Ant version 1.4. Table C.2 lists some of the optional tasks that are also publicly available for download from the Ant web site, <http://jakarta.apache.org/ant>.

Ant has been integrated with a variety of Java integrated development environments (IDE), including IBM Visual Age, JBuilder, and NetBeans. Using Ant along with your IDE makes the build process tool-independent, allowing developers working on the same project to use their favorite, separate IDE (or none at all). Ant is also integrated with the major source control systems, including the popular, open source Concurrent Versioning System (CVS).

For these reasons, Ant has become an extremely popular tool in a very short time. It is used in open source projects like W3C and ASF development activities and by commercial products. For example, BEA's WebLogic J2EE server began using Ant as of version 6, and has developed custom Ant tasks to make otherwise complicated processes easy for you. In WebLogic, you can generate the configuration files and even some of the source code for a J2EE web service simply by invoking an Ant task.

There are also many interesting open source projects based on Ant. One example is XDoclet, a custom Ant task that generates EJBs, deployment descriptors, and related objects from javadoc comments in your source code. Information on XDoclet can be found at <http://sourceforge.net/projects/xdoclet>.

We strongly recommend your use and support of this tool. Due to its central role in many Java development projects, we dedicate this appendix to an introduction of its use.

Table C.1 Built-in Ant tasks

Task name	Description
ant	Runs ant on a specific build file.
antcall	Calls another Ant task.
antstructure	Generates a (partial) DTD for an Ant build file.
apply	Executes a shell command on a specific file set.
available	Tests for the presence of a specific property within a project.
chmod	Changes permission settings on files.
condition	Sets a property if the specified condition is true.
copy	Copies files.
cvs	Accesses a CVS source code repository.
cvspass	Logs in to CVS.
delete	Removes files.
dependset	Manages arbitrary dependencies between files.
ear	Archives a file set into EAR format.
echo	Prints text to stdout.
exec	Executes a shell command.
fail	Exits the current build.
filter	Creates a token filter used to select subsets of files from a path.
fixCrLf	Adjusts a text file for local conventions.
genkey	Generates a cryptographic key.
get	Retrieves a file from a URL.
gunzip	Expands a GNU Zip archive.
gzip	Archives a file set into GNU ZIP format.
jar	Archives a file set into JAR format.

(continued on next page)

Table C.1 Built-in Ant tasks (continued)

Task name	Description
java	Invokes the java virtual machine on a class.
javac	Invokes the Java compiler.
javadoc	Invokes the javadoc utility to generate Java API documentation.
mail	Sends SMTP email.
mkdir	Creates file system directories.
move	Moves files.
parallel	Contains other tasks that are to be executed in a separate thread.
patch	Applies incremental changes to the original file(s).
pathconvert	Ensures path definitions are appropriate for the local machine.
property	Sets environment variable(s) within a project.
record	Records the build process to a file by implementing the Ant build listener interface.
replace	Substitutes text tokens across a file set.
rmic	Invokes the Java RMI compiler.
sequential	Contains other tasks that must execute in a specific order.
signjar	Signs a JAR file with a private cryptographic key.
sleep	Suspends the build process for the time period specified.
sql	Executes one or more SQL statements against a JDBC data source.
style	Applies XSLT transformations to files.
tar	Archives a file set into TAR format.
taskdef	Defines a custom Ant task for use within a project.
touch	Updates file timestamps.
tstamp	Sets date and time properties.
typedef	Defines a new Ant data type for the current project.
unjar	Expands a JAR file.
untar	Expands a TAR archive.

(continued on next page)

Table C.1 Built-in Ant tasks (continued)

Task name	Description
unwar	Expands a WAR file.
unzip	Expands a ZIP file.
uptodate	Determines if target files are older than source files.
war	Archives a file set into WAR format.
zip	Archives a file set into ZIP format.

Table C.2 A sampling of optional Ant tasks

Task name	Description
cab	Creates a MS Cabinet file archive.
cccheckin	Checks files into Rational Clear Case.
cccheckout	Checks files out of Rational Clear Case.
ccuncheckout	Release a Rational Clear Case check out.
ccupdate	Obtains the latest source from Rational Clear Case.
csc	Invokes the MS .NET C# compiler.
ddcreator	Creates EJB deployment descriptors.
depend	Manages Java class file dependencies.
ejbc	Compiles EJBs.
ejbjar	Creates EJB JAR deployment files.
ftp	Transfers files via FTP
ilasm	Provides an interface language assembler for MS.NET.
javah	Creates C/C++ header files for Java native methods.
jlink	Links Java classes and libraries from subprojects.
jpgcoverage, jpgcovmerge, jpgconvreport	Invokes utilities in the JProbe testing software suite.
junit/junitreport	Tasks for using the JUnit unit testing API.
mimemail	Sends SMTP mail with MIME attachments.

(continued on next page)

Table C.2 A sampling of optional Ant tasks (*continued*)

Task name	Description
native2ascii	Converts native file formats to ASCII with wrapped Unicode.
propertyfile	Enables unattended property file editing from within Ant.
pvcs	Interface to the PVCS source control system.
rpm	Builds Linux RPM installation files from a file set.
script	Executes a script written in any language supported by the Bean Scripting Framework (BSF).
sound	Plays a sound file at the conclusion of a build.
telnet	Enables unattended remote telnet session from within ant.
vajexport/vajimport/vajload	Implements IBM Visual Age for Java related tasks
vssccheckin, vssccheckout, vssget, vsshhistory, vsslabel	Provides MS Visual Source Safe related tasks.
wljspc/wlrun/wlstop	Executes BEA WebLogic server related tasks.
xmlvalidate	Validates XML document well-formedness.

C.1 *Installing and configuring Ant*

You can download the latest version of Ant from <http://jakarta.apache.org/ant>. Be sure to also download the optional Ant tasks. This secondary download is brief and will save you time later when you would like to invoke an optional task for the first time. Expand the Ant distribution file to a directory of your choice and set three environment variables as detailed in table C.3.

Table C.3 Ant environment variables

Environment variable	Value to set	UNIX example	MS-DOS Example
JAVA_HOME	Absolute path to your JDK installation	JAVA_HOME=/usr/local/jdk	JAVA_HOME=C:\jdk
ANT_HOME	Absolute path to your Ant installation	ANT_HOME=/usr/local/ant	ANT_HOME=C:\ant
PATH	Add Java and Ant executable directories to your system path	PATH=\$PATH:\$ANT_HOME/bin:\$JAVA_HOME/bin	PATH=%PATH%;%ANT_HOME%\bin;%JAVA_HOME%\bin

To test your installation, go to a shell and execute the command `ant` from any directory. If `ant` is installed properly, you will see the following message (assuming no file named `build.xml` is found in your current directory):

```
Buildfile: build.xml does not exist!
Build failed.
```

If you see the above message when you type `ant` at a command line, you are now ready to construct your first build file and begin using Ant.

C.2 Creating a build file

An Ant build file is an XML document with a structure represented by figure C.1. As you see, the root node of a build file is the `<project>` element. Projects can contain global properties, task definitions, dynamically constructed path variables, and targets. Targets are named groupings of tasks. A target can depend on the successful execution of other targets. A target can also invoke other targets in the course of its processing. The capabilities of Ant tasks, properties, and dynamic paths make Ant an extremely powerful and flexible tool.

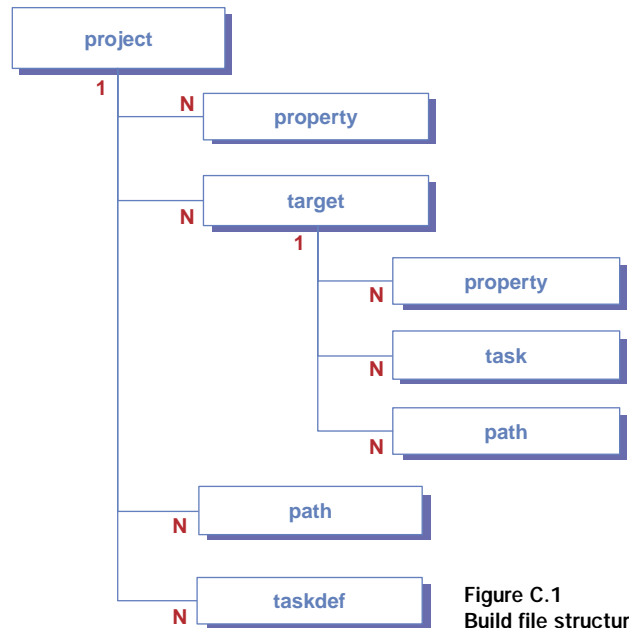


Figure C.1
Build file structure

C.2.1 *Dynamically constructed paths and file sets*

One of the key features of Ant, besides the numerous built-in and optional tasks it can perform, is its ability to dynamically construct path variables at execution time, specifically a classpath. As we mentioned in the previous section, a `<path>` is just one type of element found in an Ant build file.

Ant dynamically constructs path variables by evaluating one or more `<pathelement>` and/or `<fileset>` elements contained within a `<path>` definition. A `<pathelement>` points at a file system directory, and adds all the contents of that directory to the path being defined. A `<fileset>` is more powerful, allowing you to filter the contents of a directory being added to the path. You can specify files to be excluded either explicitly or using regular expressions.

To see how this works, let us look at an example. Suppose you define a `<path>` element in your build file as follows:

```
<path id="project.class.path">
  <pathelement path="${build.dir}"/>
  <fileset dir="lib">
    <include name="**/*.jar"/>
    <include name="**/*.zip"/>
  </fileset>
</path>
```

Ant creates a path variable with an `id` (name) of `project.class.path`. The path includes all files and directories beneath the `${build.dir}` directory (which you probably set earlier in the file via a global property). It also contains all ZIP and JAR files found in the `lib` relative directory and any subdirectories. And if you wanted to exclude a specific JAR file from the path, `deprecated.jar`, for example, you could add the following node to the fileset definition:

```
<exclude name=deprecated.jar/>
```

Once your globally declared path variable has been constructed, it can be referenced from tasks using its `id` attribute. To use the path above when compiling a Java class via the `javac` task, you would refer to it as follows:

```
<javac srcdir="${src.dir}" classpathref="project.class.path" />
```

If you have done any Java development from the command line using the JDK, you will no doubt appreciate the power and flexibility of dynamically constructed paths, and specifically the capabilities of the `<fileset>` element.

C.2.2 A Sample build file

When you download the code examples for this book, you will notice that every chapter has its own Ant build file in its base directory. Those files are very similar to the one in listing C.1. Let's examine the contents of this build file to see how Ant works.

Listing C.1 Build file contents

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

<project name="Sample" default="run" basedir="." >

  <property name="project.name" value="Ant Tutorial"/>
  <property name="ant.home" value="C:/ant"/>
  <property name="build.dir" value="${basedir}/build"/>
  <property name="src.dir" value="${basedir}/src"/>
  <property name="lib.dir" value="${basedir}/lib"/>

  <!--
    To permanently add this task to your Ant configuration,
    edit the defaults.properties file in
    ant.jar:/org/apache/tools/ant/taskdefs.
    You can then remove this declaration from the build
    file and execute the mytask task as if it were
    built into Ant.
  -->
  <!--
  <taskdef name="mytask"
    classname="MessagePrinterTask"/>

  <path id="project.class.path">
    <pathelement path="${build.dir}"/>
    <fileset dir="lib">
      <include name="**/*.jar"/>
      <include name="**/*.zip"/>
    </fileset>
  </path>

  <target name="init">
    <echo>Running "init" target...</echo>
    <tstamp/>
    <available property="build.dir.exists"
      file="${build.dir}"/>
  </target>

  <target name="prepare.build.dir"
    unless="build.dir.exists" depends="init">
```

1 Root element

Sets the global project properties

Document file with XML comments

2 Custom task definition

Adds build directory to CLASSPATH

Adds Zip and JAR files from the lib directory to the CLASSPATH

Builds a CLASSPATH variable

3 Start of target definitions

Ant task calls

This property only gets set if the build directory exists

This executes only if the unless property is not set


```

    <echo>Running "prepare.build.dir" target..."</echo>
    <echo>Creating ${build.dir}</echo>
    <mkdir dir="${build.dir}" />
  </target>

  <target name="compile" depends="init, prepare.build.dir">
    <echo>Running "compile" target..."</echo>
    <javac srcdir="${src.dir}" destdir="${build.dir}"
      classpathref="project.class.path" debug="on" />
    <echo>Built ${project.name}...</echo>
  </target>

  <target name="run" depends="compile">
    <echo>Running "run" target..."</echo>
    <java classname="MessagePrinter"
      classpathref="project.class.path" fork="yes">
      <sysproperty key="message" value="Hello, world!" />
      <arg value="message.txt" />
    </java>
  </target>

  <target name="build.task" depends="compile">
    <echo>Running "build.task" target..."</echo>
    <jar jarfile="${ant.home}/lib/myTask.jar"
      basedir="${build.dir}"
      excludes="**/MessagePrinter.class,
        **/BuildResultPrinter.class"
    />
  </target>

  <target name="run.task" depends="compile">
    <echo>Running "run.task" target..."</echo>
    <mytask message="Hello from my task!"
      file="messageFromTask.txt" />
  </target>

  <target name="build.listener" depends="compile">
    <echo>Running "build.listener" target..."</echo>
    <jar jarfile="${ant.home}/lib/myListener.jar"
      basedir="${build.dir}"
      excludes="**/MessagePrinter*.class"
    />
  </target>
</project>

```

Compiles target 4

CLASSPATH reference 5

This target invokes the JVM on the class `MessagePrinter`

6 Custom task call

- ❶ This root element of the build file defines the project name and base directory from which all other paths in the file will be built. The `default=run` attribute specifies that the target named `run` within the project will be invoked by default if no other target is specified on the command line.
- ❷ This is a task definition for the custom task we build later, in section C.3. As we will see, there are two ways to make your custom task classes available within an Ant project.
- ❸ These are the project target definitions. The `depends` attribute lists other targets that should be evaluated before this one, allowing you to build a dependency tree among targets. For example, if you invoke Ant to run the `compile` task,

```
prompt> ant compile
```

targets will be evaluated in the following order:

- 1 `init`
 - 2 `prepare.build.dir`
 - 3 `compile`
- ❹ This target compiles all Java files in the source directory and places the compiled classes into the build directory by calling the `javac` task. Note that it depends on two tasks, which are evaluated in the order in which they appear.
 - ❺ This is a call to the `javac` task, passing a reference to our dynamically constructed `CLASSPATH`.
 - ❻ This is a call to a custom task we develop ourselves in section C.3.

To use this build file, it should be saved as a file called `build.xml` in your working directory. Ant looks for a file by this name in the current directory by default. You also need to create directories named `lib` and `src` beneath your working directory. Notice that Ant will create the directory called `build` automatically if it does not exist the first time you compile your project.

Let us put Ant to work with this build file. Copy or download the source code from listing C.2 into a file named `MessagePrinter.java` in your `src` subdirectory. This class requires a system property named `message` to be set. It also accepts a command-line argument. We pass both of these to the JVM via the `java` task call in our build file:

```
<target name="run" depends="compile">
    . . .
    <java classname="MessagePrinter"
        classpathref="project.class.path" fork="yes">
        <sysproperty key="message" value="Hello, world!"/>
        <arg value="message.txt"/>
    </java>
</target>
```

After creating the source file for this class, run it using the following command:

```
prompt> ant run
```

Since run is the default target, you can also just type:

```
prompt> ant
```

You should see the message Hello, world! echoed to the console by the java task. There should also be a file named message.txt in your working directory that contains the same string. Congratulations! You are another satisfied user of Ant.

Listing C.2 A Simple class to test our Ant build file

```
import java.io.*;

public class MessagePrinter {

    public MessagePrinter() { }

    public static void main(String[] args) {
        String message = System.getProperty("message");
        System.out.println(message);
        if (args.length > 0) {
            try {
                PrintStream ps
                    = new PrintStream( new FileOutputStream(args[0]) );
                ps.println(message);
                ps.close();
            } catch (IOException ioe) {
                ioe.printStackTrace();
            }
        }
    }
}
```

Retrieves the message system property

Echoes the message to stdout

Creates a file containing the message from a command-line argument

C.3 Custom tasks

At some point in your use of Ant, you may require functionality that simply is not there. In such rare cases, you are free to develop your own task and add it to your configuration. If it is generic enough, you might share it with the rest of us. Developing a task is the right choice if there is a step in your build process that needs to happen frequently and cannot be done using a combination of existing tasks.

C.3.1 Developing the task

To see how easy it is to create a custom task, we now convert our `MessagePrinter` class from listing C.2 into an Ant task. This means extending `org.apache.tools.ant.Task` and placing our functionality in a method called `execute()` instead of `main()`. Also, parameters are passed to tags using XML attributes. To support what used to be the `message` system property and the command-line argument file name, we require two instance variables for our task class. Declaring variables and providing modifiers for them is all that is required. The code for our custom task is listing C.3.

Listing C.3 A custom task

```
import java.io.*;
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

public class MessagePrinterTask extends Task {
    private String message = null;
    private String file = null;

    public MessagePrinterTask() { super(); }

    public void execute() throws BuildException {
        if (message == null)
            throw new BuildException("You must specify a +
                \"message\" attribute to use this task.");
        System.out.println(message);
        if (file != null) {
            try {
                PrintStream ps
                    = new PrintStream(
                        new FileOutputStream(file) );
                ps.println(message);
                ps.close();
            } catch (IOException ioe) {
                ioe.printStackTrace();
            }
        }
    }
}
```

← Extends the base Ant task

← Called after attributes are set

```

    }
  }

  public void setMessage(String message) {
    this.message = message;
  }

  public void setFile(String file) {
    this.file = file;
  }
}

```

Passed attribute values
from the build file before
calling `execute()`

To compile the class defined in listing C.3, you will need to add `ant.jar` to your `lib` directory. Then execute the `build.task` target in the example build file. This will place a JAR containing the new task in Ant's `lib` directory so it can be found when you invoke it.

C.3.2 *Defining the task*

After successfully executing the `build.task` target, uncomment the `taskdef` element in the example build file. If this had been uncommented before, all your builds would have failed due to the previous absence of the JAR file containing our new task. The `taskdef` element you just uncommented makes our new task available to targets in our project. It is invoked by the name `mytask`, as seen in this target:

```

<target name="run.task" depends="compile">
  <echo>Running "run.task" target...</echo>
  <mytask message="Hello from my task!"
    file="messageFromTask.txt"/>
</target>

```

Invoking this target should produce identical results to the previous example, with the exception of a new file appearing in your working directory.

C.3.3 *Integrating the task*

To permanently add our new task to your Ant installation, you can edit the `defaults.properties` file in the Ant distribution JAR, adding its name and class name to Ant permanently. Taking the permanent approach means you need not explicitly declare the `taskdef` element in your build files anymore. See the comments in `build.xml` for more details.

C.4 Build listeners

Another extension you may wish to make to your Ant environment is to create a custom build listener.

DEFINITION A *listener* is a component that registers for callbacks with the Ant system when interesting events occur. Such events include message logging, beginning execution of targets and tasks, and ending the build process.

Common tasks for listeners are logging messages and sending notification of build problems to the configuration manager. Let us develop a simple listener to see how this aspect of Ant works.

C.4.1 Developing the listener

In this example, we develop a build listener that logs all messages produced by Ant during a build. The code for this component is shown in listing C.4. A class acting as a build listener must implement the callback methods of the `org.apache.tools.ant.BuildListener` interface. Our example listener is interested in two events; the `messageLogged` and `buildFinished` events.

Listing C.4 A build listener class

```
import java.io.*;
import java.util.*;
import org.apache.tools.ant.*;

public class BuildResultPrinter
    implements BuildListener {
    private Vector buildMessages = new Vector();

    public BuildResultPrinter() { }

    public void buildStarted(BuildEvent e) { }

    public void buildFinished(BuildEvent event) {
        Throwable e = event.getException();
        String status = (e != null) ? "failed" : "succeeded";
        String message = "Your Ant build process "
            + status + ".";

        try {
            PrintStream ps = new PrintStream(
                new FileOutputStream("build.results"));
        }
    }
}
```

Implements the `BuildListener` callback interface

Writes all logged messages to a file named `build.results`

```

        ps.println(message);
        if (buildMessages.size() > 0) {
            ps.println();
            ps.println(
                "The following messages were logged by Ant:");
            ps.println(
                "-----");
            ps.println();
            for (int i = 0; i < buildMessages.size(); i++)
                ps.println( (String) buildMessages.get(i) );
        }
        ps.close();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

public void messageLogged(BuildEvent event) { ← Collects messages
    buildMessages.add( event.getMessage() );    as they occur and
}                                               caches them

public void targetStarted(BuildEvent event) { }
public void targetFinished(BuildEvent event) { }
public void taskStarted(BuildEvent event) { }
public void taskFinished(BuildEvent event) { }
}

```

Execute the `build.listener` task in the example build file to compile and JAR this class and place it in Ant's classpath.

C.4.2 *Using the listener*

To put our new listener to work, execute the following in your working directory:

```
prompt> ant -listener BuildResultPrinter
```

You will now find a file named `build.results` in your working directory, containing all logging messages created by Ant as it evaluated the `run` target.

C.5 *Summary*

We have only scratched the surface of the capabilities and extensibility of Ant. We hope this brief introduction has been sufficient to pique your interest in this build tool. As a Java developer, especially in a team environment, learning about Ant and mastering it would be a valuable use of your time.

resources

Web-based resources mentioned in this book:

Resource topic	Web address
Ant, Cactus, Log4j, Struts, Velocity	http://jakarta.apache.org .
Cocoon, Apache Crimson, Apache FOP, Xalan, Xerces, XSLTC	http://xml.apache.org .
Concurrent Versioning System	http://www.cvshome.org .
Enhydra	http://www.enhydra.org
eXtreme Programming (XP)	http://www.extremeprogramming.org .
GMD's XQL, PDOM	http://xml.darmstadt.gmd.de/xql/
J2EE Blueprints design pattern section	http://java.sun.com/j2ee/blueprints/design_patterns/
Java XML Pack, JAXB, JAXM, JAXR, JAX-RPC	http://java.sun.com/xml .
JDOM	http://www.jdom.org
JUnit	http://www.junit.org
Mercury Interactive testing tools	http://www.mercuryinteractive.com .

(continued on next page)

Resource topic	Web address
Rational Software, Rational Unified Process (RUP)	http://www.rational.com
SOAP, SOAP specification, SOAP 1.1 with Attachments specification, XBase, XDoclet, XInclude, XLink, XML Schema, XML Signature recommended standard, XPath, XQuery, XSLT	http://www.w3.org .
This book	http://www.manning.com/gabrick
UDDI and related technologies	http://www.uddi.org
WebLogic	http://www.bea.com
Webmacro	http://www.webmacro.org
Zvon,	http://www.zvon.org

Books you may want to read:

Alur, Deepak, Dan Malks, and John Crupi. *Core J2EE Design Patterns*. New York: Prentice Hall 2001.

Beck, Kent. *Extreme Programming Explained*. Boston: Addison-Wesley, 2000.

Coulouris, George, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design*. London: Addison-Wesley, 2001.

John Davies, Rod Johnson, Cedric Buest, Tyler Jewell, Andrew Longshaw, et al. *Professional Java Server Programming J2EE 1.3 Edition*. Birmingham, UK: Wrox Press Ltd., 2001.

Fowler, Martin, and Kendall Scott. *UML Distilled*. Boston: Addison-Wesley, 2000.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Boston: Addison-Wesley, 1995.

Hatcher, Erik, and Steve Loughran. *Java Software Development with Ant*. Greenwich, CT: Manning Publications, 2002.

Husted, Ted. *Scaffolding: Developing Web Applications with Struts*. Greenwich, CT: Manning Publications, 2002.

Kay, Michael. *XSLT*. Birmingham, UK: Wrox Press Ltd., 2001.

Kruchten, Phillippe. *The Rational Unified Process: An Introduction*. New York: Addison-Wesley, 2000.

McLaughlin, Brett. *Java & XML*, 2d ed. Cambridge: O'Reilly and Associates, 2001.

Mohr, Steven, Jonathan Pinnock, and Brian Loesgen. *Professional XML*. Birmingham, UK: Wrox Press Ltd., 2000.

Mullender, Sape, ed. *Distributed Systems*. New York: Addison-Wesley, 1998.

- Seely, Scott. *SOAP*. New York: Prentice Hall, 2001.
- Vint, Danny. *XML Family of Specifications*. Greenwich, CT: Manning Publications, 2002.
- Vought, Eric, and Armin Begtrup. *Unit Testing for Java Programmers: Using JUnit and Ant*. Greenwich, CT: Manning Publications, 2002.
- Wesley, Ajamu. *Web Services Explained*. Greenwich, CT: Manning Publications, 2002.

To buy this book

A

- acceptance testing, definition 27
- Ant 26, 250–264
 - build files 255, 260
 - build listeners 263–264
 - built-in tasks 251
 - custom tasks 261–262
 - custom-built listener 263
 - dynamic filesets 256
 - dynamic paths 256
 - EJB interfaces 26
 - environment variables 254
 - installation 254–255
 - optional tasks 253
 - tasks 250
 - tutorial 235–264
- Apache 26, 196, 202
 - FOP API 191
- applets 163
- application client container 29
- application environment
 - case study 206
- application layer, defined 12
- application logic layer 13
- application tier. *See* application layer
- application, defined 11
- authentication, defined 21
- authorization, defined 21

B

- bug logs 28
- build tools 24
 - Antbuild 26

C

- caching, defined 18
- Cactus 28
- case study 204, 206, 210–225
 - detailed design 210, 215
 - implementation 215, 225
 - requirements analysis 207, 209
 - requirements definition 204–205
 - running 229
 - web service 225, 228
- client devices
 - user interfaces 159
- client/server 5
 - hybrid model 8
- code redundancy 166
- COM 154
- communication channels 4
- component interface 82
- Concurrent Versioning System (CVS) 250
- constraint-based modeling 121
- CORBA 119
- cryptography
 - asymmetrical 246
 - symmetrical 246
- CVS. *See* Concurrent Versioning System
- cXML 123

D

- database management system 18
- DCOM 119
- design tools 24
 - Rational Rose 25
 - Together Control Center 25

design validation 213
development methodologies
 and J2EE 22
 defined 22
development tools 24
 IDE 25
distributed systems 4–21
 architectures 5, 9
 challenges of 14, 21
 components of 4–5
 concepts 3
 concurrency 18
 correctness 18
 definition 2
 error handling 19
 extensibility 15
 failures 19
 flexibility 15
 heterogeneity 14
 layers 9, 12
 openness 16
 performance of 17
 scalability 17
 security 20–21
 transparency 20
 vendor independence 16
 vs. centralized 3–4
Document Object Model. *See* DOM
document type definitions. *See* XML-DTDs
DOM 45
 building with JAXP 62, 64
 JAXP 62
 versus JDOM 93
DTD. *See* XML-DTDs

E

ebXML 38, 123
 Registry 139
EDI 123
EJB 6, 153
 container 18
 object level integration 120
 software patterns 241
 testing strategy 31
Enterprise Application Archive (EAR) 33
Enterprise JavaBeans. *See* EJB
eXtensible Markup Language. *See* XQuery, XML
eXtensible Stylesheet Language Transformation.
 See XSLT
eXtreme Programming. *See* XP

F

failure transparency 20
fault tolerance, measuring 19
formatting objects 186–195
 defined 186
 formatting tree 187
FTP 11
functional testing, definition 27

H

heterogeneity 14
HTTP Unit testing 30
HTTPS 139
hybrid architecture 5
hybrid processing 8

I

IBM 138
Integrated Development Environment (IDE)
 choosing 25
integration testing, definition 27
internationalization, user interfaces. *See* localiza-
 tion

J

J2EE
 analysis tools 25
 and MVC model 164, 166
 and SOAP 125
 and systems integration 114
 application-layer pattern 239
 build tools 26
 building web services 138
 connectors 153
 data integration 115
 defect tracking tools 28
 deploying applications 33
 deployment strategies for 33, 35
 design tools 25
 development methodologies 22
 development processes 22, 29
 development tools 24–25, 29
 EJB components 31
 integrating applications 114
 message integration 117
 middleware 10–11

- J2EE (continued)
 - object integration 119
 - presentation layer 236
 - presentation tool kit 163
 - problem tracking tools 28
 - procedure integration 118
 - pure user interfaces 162, 177
 - scalability 18
 - server-side focus 8
 - Service Locator 35
 - service-layer pattern 239
 - SOAP 125
 - source code control 27
 - testing applications 29
 - testing strategies 29, 32–33
 - testing tools 27
 - testing types 27
 - user interface 162
 - web components testing 30
 - web services 141
 - J2EE Connector Architecture, purpose of 11
 - JAAS 21
 - drawbacks of 21
 - purpose of 11
 - JAF, purpose of 11
 - JAR file, archiving 33
 - Java 2 Platform, Enterprise Edition. *See* J2EE
 - Java Document Object Model. *See* JDOM
 - Java ServerPages. *See* JSP
 - JavaBeans 163
 - JAX 55–78
 - API summary 56
 - overview 55, 78
 - XML binding 56
 - XML messaging 56
 - XML parsing 56
 - XML repositories 56
 - JAXB 69–74, 124
 - overview 69, 74
 - using the objects 73
 - versus JDOM 95
 - JAXM 124, 131–138
 - asynchronous messaging 135
 - overview 76
 - synchronous messaging 131
 - JAXP 57, 66
 - and DOM 62
 - and SAX 59
 - and XSLT 63
 - configuration 58
 - for XSLT 177, 191
 - interfaces to DOM API 62
 - interfaces to SAX API 60
 - interfaces to XSLT API 63
 - packages 57
 - purpose of 11
 - XSLT 63
 - JAXR 78, 153
 - JAX-RPC 77, 124
 - JBuilder 26, 250
 - JDBC, purpose of 11
 - JDOM 66–69
 - and JAXM 131
 - building a document, example 68
 - core classes 67
 - overview 66, 69
 - use of 170
 - using 210
 - versus DOM 93
 - versus JAXB 95
 - JMS 118, 153
 - purpose of 11
 - using 211
 - JNDI 35, 153, 218
 - and transparency 20
 - purpose of 11
 - JProbe 30
 - JSP 6, 163
 - custom tags 163
 - limitations 165
 - JTA 8, 18
 - purpose of 11
 - JUnit 28
 - JUnitEE 28, 31
-
- ## L
- layers 12
 - LDAP 245
 - load balancing, defined 17
 - load testing, defined 27
 - localization, user interfaces 159
 - location transparency 20
 - Log4j 31
 - Long-Term JavaBeans Persistence 74–76
-
- ## M
- MathML 38
 - Mercury Interactive 28
 - LoadRunner 30
 - WinRunner 30

messages 4
 Microsoft 138
 middleware, purpose of 9
 mobility transparency 20
 multiple locales 159

N

.NET 138, 153–154
 NetBeans 26, 250
 network transparency 20
 non-repudiation 247
 n-tier architecture 12–14
 application logic layer 13
 data layer 13
 presentation layer 13
 services layer 14

O

object query language (OQL) 98
 omission failures 19

P

PDA's 159
 PDF format 187
 PDOM 54, 108–109
 peer processing 5, 8
 Persistent Document Object Model. *See* PDOM
 platforms 4
 presentation logic, defined 164
 problem tracking tool 24
 process failures 19
 processes 4
 proxies 17
 pUDDIng 153

R

Rational Clear Case 27, 29
 Rational Clear Quest 29
 Rational Rose 25
 Rational Unified Process 22, 204
 remote procedure calls. *See* RPC
 replication, defined 17
 RMI, and transparency 20
 RosettaNet 123
 RPC 118
 java api for 124
 RUP. *See* Rational Unified Process

S

scalability, defined 17
 security 244–247
 application components 244
 authentication models 244
 authorization models 245
 certificate authorities 244
 communication channels 244
 cryptography 246–247
 digital signatures 247
 serialization 69
 server clustering 17
 service architecture 6
 service, defined 12
 servlets 163
 filters 163, 178, 180, 212
 Simple API for XML (SAX) 44
 event handler definition 59
 Simple Object Access Protocol. *See* SOAP
 SML data storage 54
 SMTP 19, 139
 SOAP 123, 125–138
 and J2EE 125
 asynchronous messages 135
 binary attachments 129–130
 creating a message 126
 defined 49
 encoding compatibilities 154
 J2EE 125
 J2EE data types 144
 message structure 126
 message transport 128
 messaging and JAXM 131
 over HTTP 128
 request packet 127
 response packet 127
 transports 128
 software patterns 235–242
 Aggregate Entity 153
 Business Delegate 119, 131, 240
 Data Access Object 153, 241
 Model-View-Controller 163, 209
 Service Activator 211
 Service Locator 210, 239
 Singleton 210
 Value Object 240
 source code control tool 24
 SSL 247
 stylesheets. *See* XSLT
 system reliability 19

- system testing, definition 27
- systems integration 114–125
 - data level 115
 - defined 114
 - message level 117
 - object level 119
 - procedure level 118

T

- Tamino 55
- template languages, limitations 165
- testing tool 24
- thin-client application 158
- tiers 12
- Together Control Center 25

U

- UDDI 50, 123, 139, 152–153
- Universal Modeling Language (UML) 210
 - class diagrams 210
 - sequence diagrams 214
- use cases 204
- user interface
 - device types 159
 - rendering output 173
 - using XML data 170

V

- Value Object 84, 240
- Visual Age 26
- Visual Age for Java 250
- Visual Source Safe 27
- VXML 38

W

- WAR files, archiving 33
- web publishing frameworks 195–201
 - Apache Cocoon 196, 201
 - defined 195
 - Enhydra 196
 - Webmacro 196
- web services 138–154
 - architecture 6
 - defined 50, 139–140
 - in J2EE 140, 149
 - J2EE and .NET 153–154

- message style 140
- RPC-style 140
- via EJB 142
- vs. other architectures 138
- Web Services Description Language. *See* WSDL
- WebGain Studio 26
- WebLogic 229, 250
- Wireless Markup Language (WML) 159
- WSDL 51, 139, 149–152
 - example 227

X

- XBase 53
- XDoclet 250
- XInclude 52
- XLink 53
- XML
 - and relational databases 104
 - as value object 85–96
 - binary transformations 48
 - component interfaces 82–96
 - data access objects 87
 - data integration 122
 - data manipulation 51
 - databases 108
 - defined 38
 - DTDs 41–42
 - example document 40
 - inappropriate use 95–96
 - instance documents 40
 - interfaces and performance 96
 - interfaces and resources 95
 - interfaces, when not to use 95
 - Java APIs 55
 - manipulating 51–54
 - message integration 123
 - messaging 48–51
 - parsing 44–45, 61
 - persistence 96–110
 - procedure integration 124
 - querying 97–103
 - See also* XQuery, XPath
 - retrieving data 51
 - Schema definitions 42
 - storing data 54, 103
 - systems integration 122–125
 - transforming 46–48
 - translation technologies 46
 - user interface 170, 177, 201

- XML (continued)
 - uses 38
 - validation of 41–44
 - validation technologies 41
 - value objects, implementing 87
 - web publishing 195
 - writing a JavaBean 74
- XML Databases 55
- XML Query Language. *See* XQuery, XQL
- XML Schema definition 42–44
 - example 43
- XML Signature 130
- XP 23, 204
- XPath 51
- XPointer 52
- XQL 54, 101–103
- XQuery 54–101, 122
- XQueryX 98
- XSD. *See* XML Schema Definition
- XSLT 46–48, 122
 - binary transformations 48, 186–195
 - building stylesheets 183–185
 - client-side 201
 - stylesheet selection 179
 - user interfaces 177–195
 - web processing flow 178

More benefits ...

To learn about Manning's other fine Java books,
go to www.manning.com/java.html

Special offer:

To buy this book online at a great price and with the fastest delivery,
go to www.manning.com/gabrick/specialorder.html