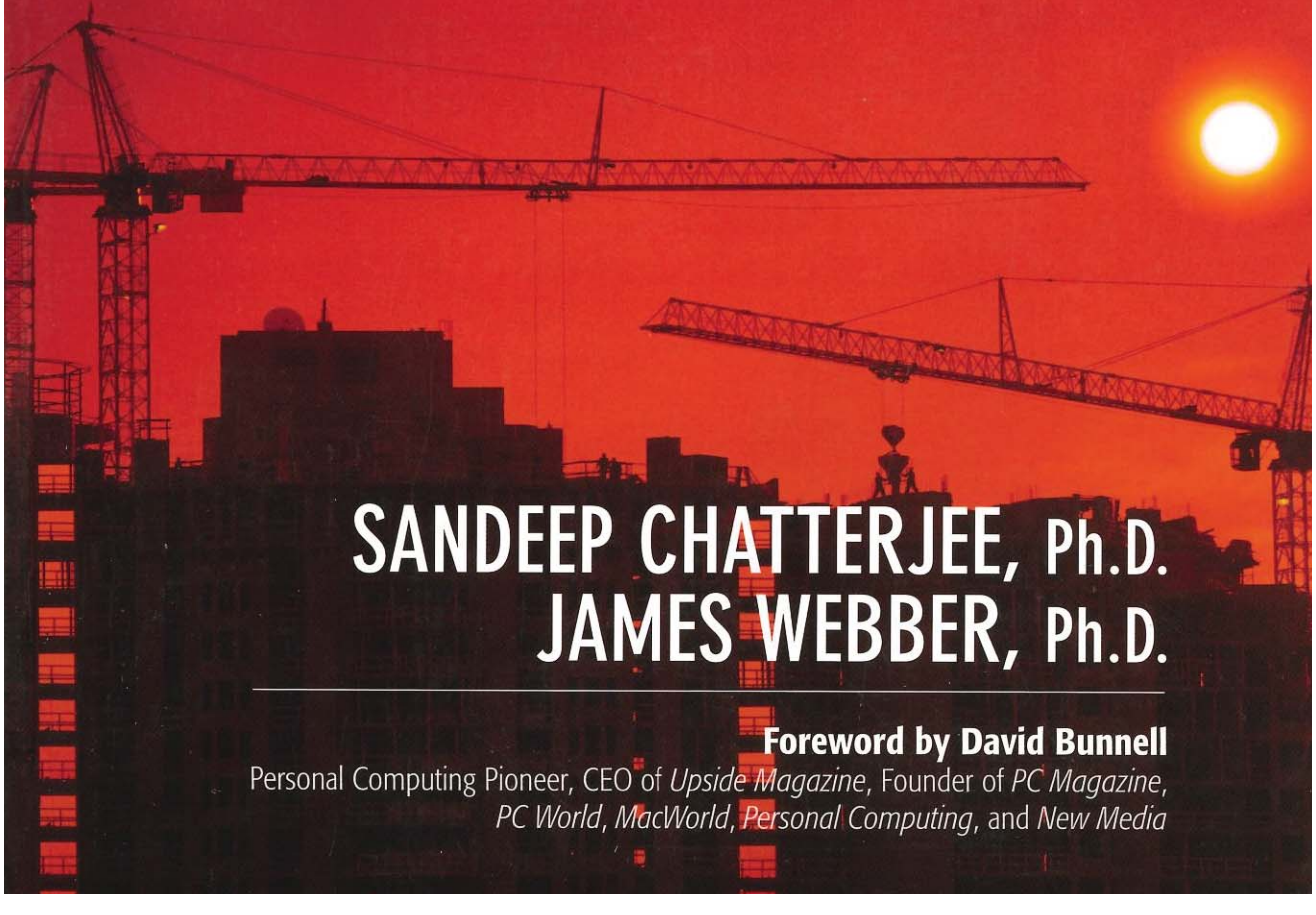
publishing partner hp invent

"...[an] invaluable resource for developers and managers who want to look past the standard stock quote service."

—Eric Newcomer, CTO Iona

Developing Enterprise Web Services

An Architect's Guide



SANDEEP CHATTERJEE, Ph.D.
JAMES WEBBER, Ph.D.

Foreword by David Bunnell
Personal Computing Pioneer, CEO of *Upside Magazine*, Founder of *PC Magazine*, *PC World*, *MacWorld*, *Personal Computing*, and *New Media*

Developing Enterprise Web Services

An Architect's Guide

Sandeep Chatterjee, Ph.D.
James Webber, Ph.D.

invent

www.hp.com/hpbooks

PRENTICE HALL PTR

PEARSON EDUCATION
PRENTICE HALL PROFESSIONAL TECHNICAL REFERENCE
UPPER SADDLE RIVER, NJ 07458
WWW.PHPTR.COM

Library of Congress Cataloging-in-Publication Data

A CIP catalog record for this book can be obtained from the Library of Congress.

Editorial/production supervision: *Mary Sudul*
Cover design director: *Jerry Votta*
Cover design: *DesignSource*
Manufacturing manager: *Maura Zaldivar*
Acquisitions editor: *Jill Harry*
Editorial assistant: *Brenda Mulligan*
Marketing manager: *Dan DePasquale*
Publisher, HP Books: *Mark Stouse*
Manager and Associate Publisher, HP Books: *Victoria Brandow*

PRENTICE HALL PTR

© 2004 Hewlett-Packard Corp.
Published by Prentice Hall PTR
Pearson Education, Inc.
Upper Saddle River, New Jersey 07458

Printed in the United States of America
2nd Printing

ISBN 0-13-140160-2

Pearson Education LTD.
Pearson Education Australia PTY, Limited
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd.
Pearson Education Canada, Ltd.
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education — Japan
Pearson Education Malaysia, Pte. Ltd.

FOREWORD

The singing workmen shape and set and join
Their frail new mansion's stuccoed cove and quoin
With no apparent sense that years abrade...
—Thomas Hardy, Rome: Building a New Street in the Ancient Quarter, 1887

OK, Rome wasn't built in a day, but once they got the sucker up and running, it was magnificent and, hey, it's still there and functioning quite nicely. Having first heard about Web services toward the end of the last century, I would have thought by now they would be ubiquitous. At this point in time, I should be able to replace My Yahoo with a personalized Web services portal uniquely suited to my quixotic needs and desires. Years ago, I started planning this portal when I heard Bill Gates waxing poetically about Hurricane—a.k.a. "My Services"—which was Microsoft's vision of creating a suite of personalized Web services for early adopters like me. Unfortunately, Microsoft abandoned this effort when critics complained it was really an insidious plot to own people's personal information.

Mostly by pure dumb luck, I've been at the forefront of technology for most of my life. As a young man just out of college, I was working in Albuquerque, New Mexico, at a small company called MITS which, with a little help from a then 19-year old Bill Gates and his buddy Paul Allen, started the personal computing revolution. Taking advantage of this happy situation, I leveraged my background in media to launch a magazine called *Personal Computing*. These experiences led me to found a number of other magazines including *PC Magazine*, *PC World*, *Macworld*, *Publish*, *NewMedia*, and *BioWorld*. Most recently I was CEO and Editor of *Upside Media, Inc.*

Throughout the years, I have been fortunate to have had a first-hand involvement in the evolution of many revolutionary new innovations, including the first personal computer (Altair), the first portal computer (Osborne), the first spreadsheet (VisiCalc), the Macintosh Computer, Multi-Media, the Internet, and even Biotechnology.

To say that I have seen my share of "paradigm shifts" is an understatement. Technology innovation has been all around me. Who would have thought that a couple of guys in a small company in Albuquerque would start what would later become the PC revolution? Shouldn't it have come out of IBM or Xerox or some other big technology company? That's exactly the rub. Innovative ideas don't always come from the big companies, sometimes they spring out from the big guys and at other times they spring out from the little guys.

Based on all the above, I am completely convinced that Web services will level the playing field between the little guy and the big guy as no technology has ever done before. Thanks to this new revolution, the mom-and-pop company down the street can market their innovative software and services to the neighborhood, to the global masses, as well as to the largest companies in the world. But, we're not only talking about the new and interesting. Even the most mundane and boring is supported. The procurement system of the mom-and-pop company can seamlessly interface with the billing system of a global multinational company and here's where things get really interesting. The systems of the multinational can also interface with the systems of the mom-and-pop company. The most innovative new systems to the most boring, existing tasks are all available on an anybody-to-anybody basis. This will ultimately happen but like many great technologies, it will require a lot of work and patience before the dream is truly realized.

As Sandeep Chatterjee and James Webber so eloquently and clearly explain in this book, real world Web services and Web services-based applications can't simply be put together in a haphazard manner by merely reading through one of the Web services technology specifications. You need to be familiar with these standards and they are extremely important, but they only represent the basic building blocks. To "architect" and construct world-class enterprise services, developers need a much deeper knowledge of a number of different standards and tools plus their "inter-relationships" and best practices for use.

Web services are small segments of larger applications and as such, quality-of-service issues loom large if they are to be truly useful and scalable. When building them, you have to factor in such considerations as: *Availability* (how often is the service available for consumption); Accessibility (can it serve a client's request now); *Performance* (how long does it take to respond); *Compliance* (is it really "standard"); *Security* (is it safe to interact with this service); *Energy* (suitable for mobile apps); and *Reliability* (how often does it fail). Like building Rome, building Web services gets complicated fast.

So how do you architect an application to be reliable if some of the Web services you are depending on become unavailable? Can an application be written to seamlessly scale to support new Web services from an expanding group of strategic partners? What about transactional guarantees or atomic coordination between multiple, independent services? And can you accomplish your design goal and still provide adequate safeguards for corporate and individual information and intellectual property?

I imagine that the software world would have given up in disgust by now, moved on to some new paradigm, except for two factors. The first is that all the major software companies are committed to Web services to the tune of several billion dollars, and the second is that Web services are, gosh darn-it, actually revolutionary. They are so revolutionary they represent a whole new amazing way of doing business, which will transform the software industry *forever* and change the very nature of the corporate IT department, thrusting it into the heart of strategic thinking.

Web services build on and extend the Web application model by allowing any client application to access and use its capabilities. By implementing capabilities that are available to other applications (or even other Web services) via industry standard network and application interfaces and protocols, Web services represent reusable software building blocks that are URL addressable. We're talking here about a concept called "anybody-to-anybody" communications—quoting from this book, "a person who implements a Web service can be almost one hundred percent certain that anybody else can communicate with and use the service."

Chatterjee and Webber aren't so concerned, however, about Web services for the masses. They tackle a much more difficult topic, which is Web services for enterprises. These are services that have to be totally reliable, absolutely secure and extremely functional. Referring back to the "building Rome" analogy, these guys aren't really talking about building foot paths or neighborhood streets, rather they are more interested in the avenues, aqueducts, and other major arteries that seamlessly and safely interconnect the Porticus of Gaius to the Forum of Caesar—the House of the Vestal Virgins to the Temple of Saturn, and back again. They are talking about the communication and transportation systems that made Rome the most magnificent functioning city of the Ancient World.

In today's global marketplace, world class enterprises need to interconnect with their customers and partners internationally using legacy systems that are mostly incompatible with each other, and they need to do this relatively fast and as inexpensive as possible. Web services provide the solution but not without overcoming some fairly difficult obstacles.

In the Web services world, nothing is as simple as it may seem. Take transactions, for example. Transactions are the bedrock on which B2B interactions rise or fall, they are a fundamental abstraction or requirement for what we sometimes refer to as "fault-tolerant computing." In a lucid and detailed style, the authors point out that the choices for transactions are scarce and, in fact, the OASIS Business Transaction Protocol (or simply BTP) is the "only Web services transaction protocol with implementation we can use today." They explain BTP and how to implement it, but just in case you get in over your head, they also suggest that unless you have specialist knowledge in this area, you should give "serious consideration" to buying or outsourcing it.

As with transactions, this book goes into great detail to describe the Web services technologies and standards that can be used in the real world today. These address the most challenging enterprise requirements including conversations, workflow, security, the challenges inherent in the development of mobile systems, how to build user-facing portals by aggregating back-end Web services, and how to manage an ever growing number and type of Web services within the enterprise. But more than this, the authors tell you in a concluding section filled with source code and a step-by-step guide how to put this together. You'll learn how to actually develop a

Web service application and deploy it onto the Tomcat application server and the Axis SOAP server (both freely available).

The ambitious goal of *Developing Enterprise Web Services: An Architect's Guide* is to give readers a "thorough understanding" of the steps necessary to build and deploy Web services and client applications that meet enterprise requirements. This is a lofty goal indeed, and you'll want to spend some serious time going through all the clear and concise content that the authors have spent well over a year developing. I found it really amazing.

Fortunately, with the publication of this book, the Web services vision is about to take a giant leap forward. We are building our "Rome" and the end is finally in sight. Chatterjee and Webber, drawing on their own impressive experiences building Web services, painstakingly provide their readers with concise, yet thorough understanding of the most important issues and their solutions. They unabashedly recommend best practices in application architectures, put key technologies together and show their readers step-by-step how to build world-class, enterprise Web services-based e-business applications. And darn it, it's about time we had a book like this!

David Bunnell
Berkeley, California
September 2003

CHAPTER 1

Introduction

Web services technologies are fundamentally changing the software industry, making the role of enterprise IT organizations more strategic, and recasting the software vendor-consumer relationship. Web services are also being hailed by CEOs, CIOs, and CTOs as the next-generation vehicle for driving topline growth and controlling bottom lines. But, simply jumping on the Web services bandwagon won't lead to corporate success. Web services are simply a platform; how companies implement a solution using this new technology determines their success and ultimately their return on investment (ROI). In this book, we take a no-nonsense, strategic view of developing enterprise Web services and applications: looking at where the technologies are, where they are going and how companies need to architect their own Web services solutions to not get left behind.

Web services platforms provide the functionality to build and interact with distributed applications by sending eXtensible Markup Language (XML) messages. Additional technology layers are constantly emerging, others are being refined, and still others are being discarded. The platform is essentially a moving target.

To stay on the leading edge, companies are building and deploying their applications while work on the underlying platform continues. And, as with any industry standard initiatives which require building consensus, the Web services platform will remain a work in progress for some time.

How can you build any meaningful application, let alone mission-critical enterprise applications, on such a platform? If you are a developer or an architect charged with building Web services or applications that consume Web services, you have to know where the platform is today, and where it is going. Otherwise, the endless pit of application rewrite and maintenance overhead will far outweigh any benefits that can be garnered from this promising new technology.

Real world, enterprise Web services and applications cannot be developed by simply reading through the Simple Object Access Protocol (SOAP) or the Web Services Description Language (WSDL) specifications. Developers must understand a number of different standards and technologies, and more importantly, their inter-relationships as well as best practices for their use.

Consider an e-business application that requires interaction between multiple partner Web services. Understanding SOAP and WSDL gives developers the ability to write Web services and consume them within their application. But, how must the application be architected to be reliable in case some Web services become unavailable? How can an application be written to seamlessly scale and support new Web services from a growing list of strategic partner companies? What are the best practices for developing mobile Web service applications, and how can individual Web services be created to support quality-of-service (QoS)? How can transactional guarantees or atomic coordination between multiple, independent Web services be supported by applications? And, how can all of this be done securely so that corporate and individual information and intellectual property are safeguarded?

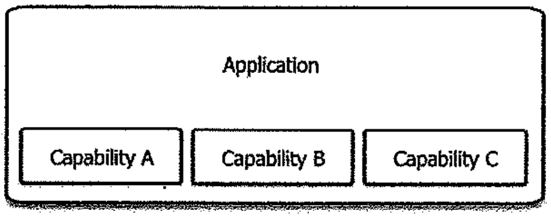
In this book, we focus on how to develop Web services and applications within real world enterprise environments. We describe not only the vanilla Web services platform consisting of SOAP, WSDL, and UDDI (Universal Description, Discovery and Integration), but also build on this to include the other technologies, standards, and emerging standards that provide support for transactions, security and authentication, mobile and wireless, quality-of-service, conversations, workflow, interactive applications and portals, as well as systems management.

We discuss the opportunities represented by Web services and, more importantly, describe best practices and architectural patterns for building enterprise systems that position you and your organization to most fully leverage those opportunities. We do not summarize any one Web services standard, but instead provide a sufficiently thorough discussion of all of the critical technologies and standards, as well as their inter-relationships, that are necessary for building enterprise Web services and applications. Our focus is on developing enterprise Web services and applications based on industry standard Web services technologies, not on summarizing standards.

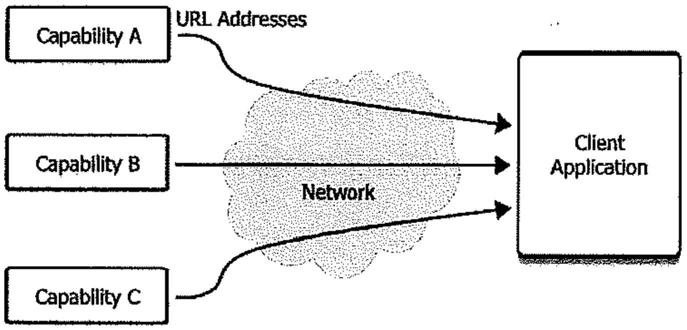
Let's get started by reviewing what Web services are and why they are important.

What Are Web Services?

Web services represent a new architectural paradigm for applications. Web services implement capabilities that are available to other applications (or even other Web services) via industry standard network and application interfaces and protocols. An application can use the capabilities of a Web service by simply invoking it across a network without having to integrate it. As such, Web services represent reusable software building blocks that are URL addressable. The architectural differences between monolithic, integrated applications and Web services-based applications are depicted in Figure 1-1.



(a) Monolithic application with integrated capabilities A,B, and C.



(b) Client application invoking remote Web services for capabilities A, B, and C.

Figure 1-1 The architectural differences between (a) a monolithic application with integrated capabilities, and (b) a distributed application using Web services-based capabilities.

The capabilities provided by a Web service can fall into a variety of categories, including:

- Functions, such as a routine for calculating the integral square root of a number.
- Data, such as fetching the quantity of a particular widget a vendor has on hand.
- Business processes, such as accepting an order for a widget, shipping the desired quantity of widgets and sending an invoice.

Some of these capabilities are difficult or impractical to integrate within third-party applications. When these capabilities are exposed as Web services, they can be loosely coupled together, thereby achieving the benefits of integration without incurring the difficulties thereof.

Web services expose their capabilities to client applications, not their implementations. This allows Web services to be implemented in any language and on any platform and still be compatible with all client applications.

Each building block (Web service) is self-contained. It describes its own capabilities, publishes its own programmatic interface and implements its own functionality that is available as a hosted service. The business logic of the Web service runs on a remote machine that is accessible by other applications through a network. The client application simply invokes the functionality of a Web service by sending it messages, receives return messages from the Web service and then uses the results within the application. Since there is no need to integrate the Web service within the client application into a single monolithic block, development and testing times, maintenance costs, and overall errors are thereby reduced.

Assume you want to build a simple calculator application that determines the appreciation in stock price for any company given its corporate name and the date the stock was originally purchased. The application must do the following:

- Determine the stock ticker symbol for the company based on the company name.
- Determine the latest price of the stock based on the ticker symbol.
- Determine the historical price of the stock for the given date based on the ticker symbol.
- Calculate the difference between the two stock prices and present it to the user.

This seemingly trivial application is in fact enormously complex. Right from the get go there are problems. We have to build a database with the names of all the companies in the country and their associated stock ticker symbol. More importantly, we must maintain this database as companies are newly listed, become delisted, change their names or their ticker symbol, or merge. To access the real-time price of a stock, we must have a relationship with a financial or brokerage firm. The legal complexities and hassles in architecting such a relationship is bad enough, not to mention the IT infrastructure that must also be put into place.

Unless you work for a brokerage firm or are in the business of maintaining stock information, the time and costs necessary to build the infrastructure necessary to support the stock appreciation calculator are enormous and, in most cases, prohibitively so. Until a brokerage firm itself decided to provide such a calculator, customers would have to make do without it.

Web services simplify and in many ways eliminate the need to build for yourself the support infrastructure—both legal and technical. The calculator can be developed by simply passing messages between the calculator application and the appropriate set of Web services. Figure 1-2 graphically depicts the flow of messages, and the fundamental architecture of a Web services-based stock price appreciation calculator.

Messages are sent between the calculator application and the following three Web services:

- `StockTickerNameToSymbolConverter`, which accepts a company's name and provides the ticker tape symbol.
- `RealTimeStockQuoteLookup`, which provides the latest price of a stock based on its ticker tape symbol.

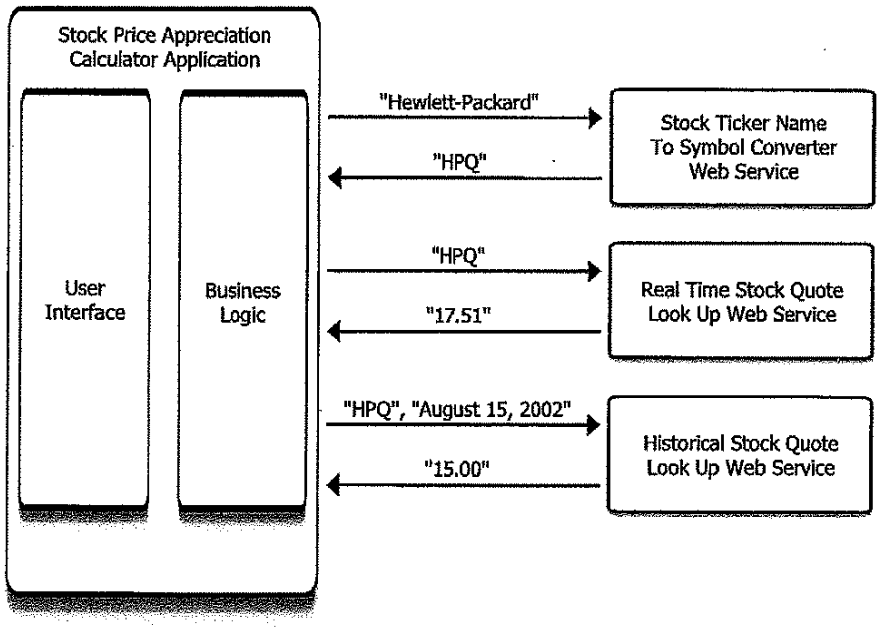

Stock Price Appreciation Calculator Application
User Interface
Business Logic
"Hewlett-Packard"
"HPQ"
Stock Ticker Name To Symbol Converter Web Service
"HPQ"
"17.51"
Real Time Stock Quote Look Up Web Service
"HPQ", "August 15, 2002"
"15.00"
Historical Stock Quote Look Up Web Service

Figure 1-2 Sending and receiving Web service messages to build a stock price appreciation calculator.

- `HistoricalStockQuoteLookup`, which provides the historical price of a stock based on its ticker tape symbol and the desired date.

Since each of these three Web services is provided, hosted, and managed by another company, the developer of the calculator application has only to focus on his key insight or contribution alone. Complex, domain-specific issues such as the fact that Hewlett-Packard's ticker tape symbol was HWP and only recently became HPQ are (or should be) handled by the Web services directly. Using these three Web services, the application can easily determine the stock price appreciation for Hewlett-Packard from August 15, 2002, to be $17.51 - $15.00 = $2.51. Based on the data from the Web services, the calculator application can provide further analysis, such as the percentage appreciation, and present all of the information in an easy-to-understand, graphical manner.

Assuming the required capabilities exist and are available as Web services, developers can focus on their unique value-added piece and utilize third-party Web services for the remainder of the functionality. The benefits of using Web services are clear:

- Dramatically cut application development costs by focusing on your own value-added contribution and using third-party Web services for everything else.

- Integrate both data and business processes with market constituents and business partners that have desired domain expertise or capabilities.
- Reduce or eliminate many errors born out of complex and large monolithic applications.
- Simplify application maintenance and customization by segmenting an application into the client application and each of its consumed Web services.
- Significantly reduce time-to-market.

As we take this idea further, and more and more companies expose some of their internal capabilities as Web services, the real value of Web services lies in the *composition* of a set of Web services. Consider the following two companies. One is a traffic service company that monitors automobile traffic on major roads and highways and predicts expected travel times. The second is a taxi reservation service company that allows customers to reserve taxis for pickup at a specified location and time. Each of these companies and their products are compelling in and of themselves. However, if these companies exposed their capabilities as Web services, these services can be *composed* together into a single, more compelling and useful service—either by one of these two companies themselves or by a third company.

As an example, consider taking a taxi to the airport before catching a flight for a meeting. By leveraging the capabilities of both companies through their respective Web services, a traveler can reserve a taxi and rest assured that if an accident or other traffic conditions cause an unexpected increase in her travel time, the taxi reservation can be held and an alert sent to the traveler advising her of the updated taxi schedule as well as the traffic situation that caused the change. By simply and intelligently combining the individual services of the two companies, we are able to create a more compelling and useful service for travelers. The composition of Web services from different enterprises is depicted in Figure 1-3. The technologies that form the foundations of Web services are SOAP, WSDL, and UDDI.

SOAP

Simple Object Access Protocol (SOAP) is an XML-based mechanism for exchanging information between applications within a distributed environment. This information exchange mechanism can be used to send messages between applications and, more specifically, can be used to implement remote procedure calls (RPCs). RPCs allow one application to invoke and use a procedure (or capability) of another, possibly remote, application.

SOAP does not specify any application implementation or programming model. Instead, it provides a mechanism for expressing application semantics that can be understood by applications no matter how they are implemented. Accordingly, SOAP is application language- and platform-independent. SOAP is typically used in conjunction with HTTP, which supports easy traversal of firewalls and is sufficiently lightweight to be used within mobile and wireless environments.

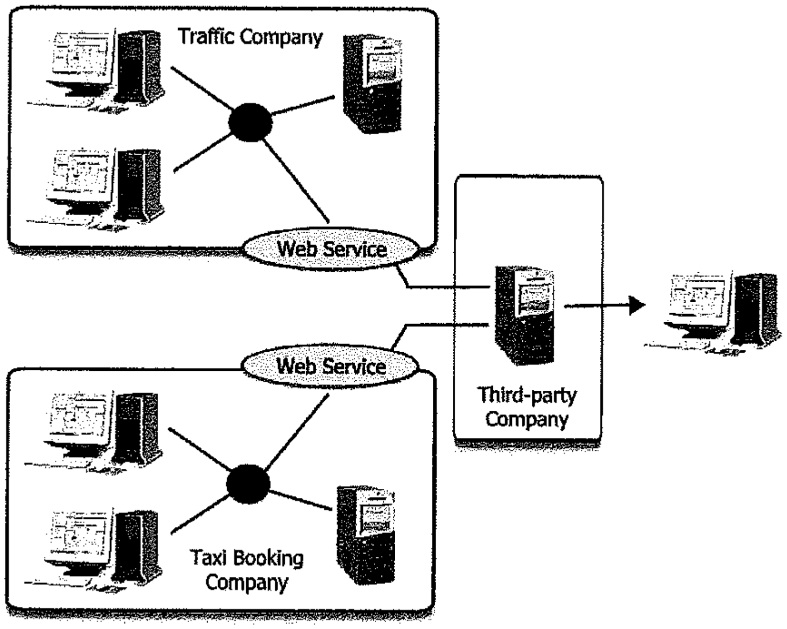


Figure 1-3 Composing together services exposed by multiple corporations to create a separate service offering.

WSDL

Web Services Description Language (WSDL) is an XML-based language for describing Web services. Through a WSDL description, a client application can determine the location of the remote Web service, the functions it implements, as well as how to access and use each function. After parsing a WSDL description, a client application can appropriately format a SOAP request and dispatch it to the location of the Web service.

WSDL descriptions go hand-in-hand with the development of a new Web service and are created by the producer of the service. WSDL files (or pointers thereto) are typically stored in registries that can be searched by potential users to locate Web service implementations of desired capabilities.

UDDI

Universal Description, Discovery, and Integration (UDDI) is a specification for a registry of information for Web services. UDDI defines a means to publish and, more importantly, discover (or search for) information about Web services, including WSDL files.

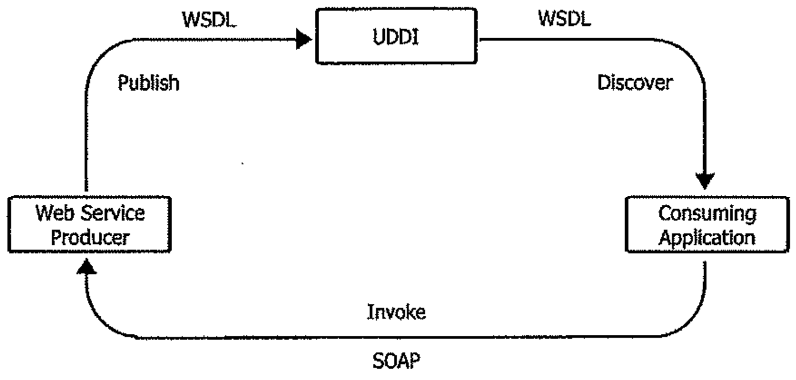


Figure 1-4 The relationships between SOAP, WSDL, and UDDI.

After browsing through an UDDI registry for information about available Web services, the WSDL for the selected service can be parsed, and an appropriate SOAP message can be sent to the service. Figure 1-4 graphically illustrates the relationships between SOAP, WSDL, and UDDI.

Now that we have a glimpse into what Web services are and how they can be used to build interesting applications and systems, we next discuss why this new technology is important.

Why Web Services Are Important

Web services represent a new paradigm in application architecture and development. The importance of Web services is not that they are new, but that this new technology addresses the needs of application development. To understand this new paradigm, let us first look at the application paradigm that preceded Web services—Web applications.

The Evolution of Web Applications

Web applications are applications that are available via the World Wide Web (Web) and allow any user anywhere in the world access to its capabilities. This is in contrast to older client-server applications in which only dedicated clients could access the applications residing on the server. Web applications grew the user base from just a few hundred client machines accessing a client-server application, to millions of users across the Web accessing a Web application.

The Web opened up the floodgates to Web applications by allowing users to simply specify a URL within a Web browser. Web applications also increased the difficulty of developing applications because a Web application client (a PC browser) has no knowledge of the application's communication requirements or underlying systems. Industry standard technologies such as HTTP and HTML were used to bridge this gap between Web application clients and the Web applications themselves. Application servers and other middleware emerged to reduce the complexities of building Web apps while still allowing pervasive access to each Web application.

Web services build on and extend the Web application model. Web applications allow any Web browser to access its functionality, with the application user interface presented through the browser. Web services take this a step further and allow any client application to access and use its capabilities.

A Web application allows universal user access to its capabilities by supporting industry standard interfaces to its user interface. They do not allow extending or adding to their capabilities through programmatic access. To leverage the functionality of a Web application and build on it, complex and often unreliable techniques, such as screen scraping, must be used. Web services address this issue by allowing programmatic access to the Web services' capabilities using industry standard interfaces and protocols. The evolution of Web applications to Web services is shown in Figure 1-5.

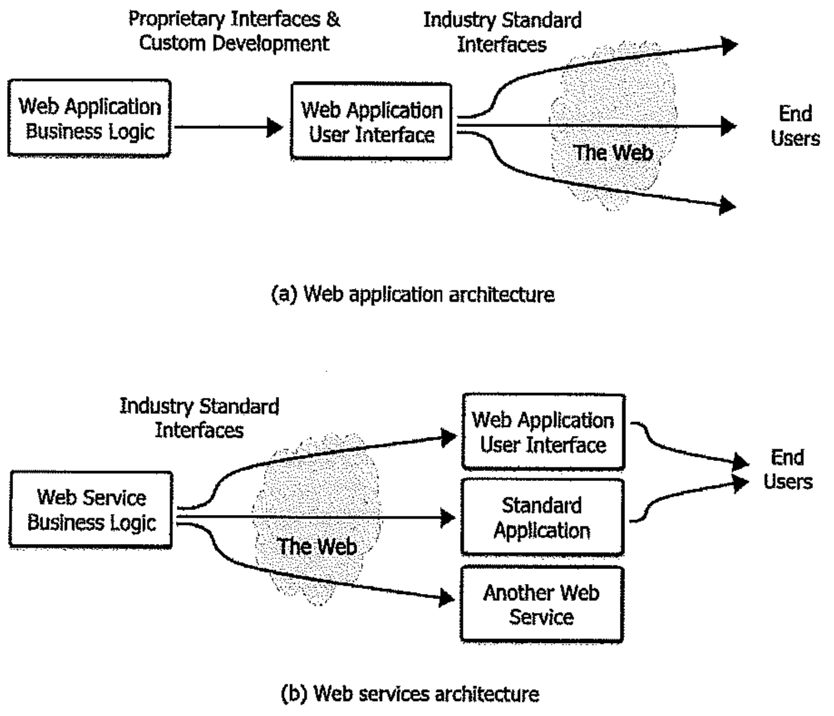

Proprietary Interfaces & Custom Development
Industry Standard Interfaces
Web Application Business Logic
Web Application User Interface
The Web
End Users
(a) Web application architecture
Industry Standard Interfaces
Web Service Business Logic
The Web
Web Application User Interface
Standard Application
Another Web Service
End Users
(b) Web services architecture

Figure 1-5 Evolution of Web applications to Web services and key architectural differences.

Web services advocate a services-oriented architecture for applications in which a software component provides its functionality as a service that can be leveraged by other software components. Such a service model abstracts away many complex issues that arise from software component integration, including platform compatibility, testing, and maintenance.

Since Web service clients do not have information necessary to communicate with a Web service, a set of standards is necessary to allow any-to-any communications. Web service standards build on previous standards for communications and data representation, such as HTTP and HTML.

The key enabler for Web services is XML. Although HTML and XML are similar in that both are human-readable markup languages, HTML is for presentation markup while XML is for semantic markup. This critical attribute of XML supports expressing application and functional semantics in a platform-independent manner that enables any-to-any information exchange.

Some argue that Web services are nothing new; they are simply the latest incarnation of distributed computing. In some sense that may be true, but what is it about Web services that is driving the incredible buzz? Why are entrepreneurs, CEOs of established companies, and industry analysts excited about this technology? In the next section, we see that Web services are not just another distributed computing platform.

Not Just Another Distributed Computing Platform

Web services are indeed a technology for distributed computing and there is one critical distinction between Web services and distributed computing technologies that have come before. A person who implements a Web service can be almost one hundred percent certain that anybody else can communicate with and use the service. The breakthrough of Web services is precisely the anybody-to-anybody communications that it enables. The confidence level Web services engender in its developers is similar to that of HTML Web pages. The developer of an HTML page is certain that anybody with a browser can view the Web page.

Web services grew out of a need for a distributed computing application environment that was not as difficult to deploy to as the Common Object Request Broker Architecture (CORBA) or Microsoft's Distributed Component Object Model (DCOM), and also offered greater interoperability. Both CORBA and DCOM aimed to provide a distributed computing environment across heterogeneous environments. Unfortunately, neither supported environments or technologies that were sufficiently far-reaching to enable heterogeneous communications at the anybody-to-anybody scale.

In a sense, Web services sacrifice the richness of capabilities that are provided by previous distributed computing environments, which are necessary to a small group of all applications, for a much simpler and more ubiquitous solution that is applicable for the vast majority of applications. This is not to say that Web services place restrictions on their use. Additional capabilities can be layered on top of the Web services platform to address varying needs.

Applications that are exposed as Web services have a large base of other applications (that are also exposed as Web services) with which to communicate. Since they are based on simple and open industry standards (or de facto standards), Web services make significant inroads toward ubiquitous interoperability. Interoperability here is on the scale of the Web or the Internet, not just a group or organization.

Based on industry standards and supporting anybody-to-anybody interoperability, Web services are poised to be the platform that delivers on the needs of e-businesses. All companies interact with other companies in the course of conducting their businesses. Manufacturing companies interact with component suppliers, distributors interact with manufacturing companies, retailers interact with distributors, and so on. Initially, these interactions were manual, conducted by mail, phone, and fax.

Web applications allowed companies to interact with one another by exposing some of their capabilities and business processes to others on the Web. But, most of the time, this still required a human being interacting with the Web application on the other side. Web services remove the need for constant human intervention while companies interact by enabling programmatic conversations between applications.

By removing this barrier to e-business interactions, Web services enable new business relationships as well as more fluid relationships that can be configured and reconfigured on-the-fly. Although Web services offer numerous benefits, they also present many challenges and risks within traditional enterprise environments. We discuss Web services and how they fit within enterprises next.

Web Services and Enterprises

On the surface, Web services appear to be a risky proposition for enterprises. Why will IT organizations that have demanded full control over all aspects of enterprise applications adopt a distributed and shared software architecture that moves administrative control over various parts of applications outside of the enterprise firewall? The runtime characteristics of Web services-based applications will have critical dependencies on remotely hosted and remotely managed external businesses. This is a severe departure from the centrally controlled as well as the guaranteed predictability and reliability that have become the hallmarks of enterprise software and the IT organizations that manage them.

The reasons for this break are clear. Web services enable the flow of data and business processes between business partners—between enterprises as well as between multiple organizations or groups within an enterprise—to a degree that have not been possible before. Businesses that could not previously communicate and applications that could not previously interoperate can now do so.

Web services enable companies to drive topline growth by integrating together different services and introduce new revenue-generating services. At the same time, Web services simplify integration, reducing time-to-market and costs, as well as support operational efficiencies that streamline the bottom line.

The potential benefits of Web services are enormous. The risks are equally great, if not greater. Enterprise IT organizations will find themselves in the middle, responsible for reconciling the benefits with the risks of adopting Web services within the enterprise.

IT organizations, in an effort to gain a controlling foothold over risky and potentially harmful Web services traffic, will insist on controlling which Web services applications interact with one another. A misbehaving Web service will simply be cut off from interacting with any enterprise applications; such cut offs may even be preemptive if there is a history of problems or a perception of a threat.

To accomplish this, IT will take on a more strategic role within organizations and align itself more closely with individual business units. Critical decisions by business units, such as the partners from which to source components, will have to be cleared by IT if those partners' Web services will interact with the applications or Web services of the company.

This will have major ramifications for enterprise application architectures. Enterprise applications will support dynamic and swappable Web services—hardwired Web service invocations will no longer suffice. Moreover, IT will use management environments to deploy enterprise-wide policies for Web services that will monitor and strictly enforce the Web services that applications can use.

There is no doubt that the uptake of Web services within the enterprise will require changes. Many of these changes will be to established procedures and existing policies that have been supported by years of experience and billions of dollars. Nonetheless, the potential benefits—both financial and strategic—to adopting Web services are sufficiently large to justify such changes.

Moving Forward

As organizations transition from researching Web services technologies and building internal prototypes to early-adopter deployments, and then eventually to mainstream adoption of Web services, the key differentiator and requirement is that these applications and systems are appropriate for "real world" deployment and usage. Some of the early prototypes built using Web services were in many ways toys. All of the Web services and client applications run on a single machine, hosted by the same application server, in a fully controlled environment. Many of these services and applications that once worked as prototype systems will no doubt break—some immediately, while others may take more time to break (which is much worse).

The next few years will see Web services and applications become hardened and ready for "real world" deployment. The real world is indeed a cold and hard place. Web services run remotely, sometimes go down and become unavailable, evolve into multiple versions, as well as encounter variances in network bandwidth and quality. Moreover, politics and competitive issues between organizations will result in unexpected outages and behaviors along critical dependencies within applications.

Already we see many standards bodies that have been convened to address these and other issues. Some of the technologies that are being developed to address these needs will eventually be automatic, transparent to developers as existing infrastructure and tools, such as middleware and IDEs, and incorporate the technologies. Nonetheless, architects and developers will need to have a keen understanding of these issues and technologies to develop enterprise-class Web services and applications.

In this book, we look at the Web services platform—where it is now and where it is going—with an eye toward developing robust enterprise Web services and applications. In the first of the three sections of this book, we begin by describing the core technologies that make up the Web services platform. These are XML, SOAP, WSDL, and UDDI. This platform provides a distributed computing environment based on standard interfaces and protocols, but it does not implement all of the capabilities necessary for implementing enterprise systems.

In the second part of this book, we look at some of the standards and emerging technologies that, once layered on top of the vanilla Web services platform, address some of the critical requirements of enterprise systems. These technologies include support for transactions, security and authentication, conversations, workflow, quality of service, mobile and wireless, services and systems management, as well as interactive applications and Web portals.

In the third part of the book, with both the vanilla Web services platform as well as some of the critical advanced technologies and standards under our belt, we take an in-depth look and provide step-by-step instructions for building an enterprise application using Web services. Addressing one of the biggest pain points in business processes today, we develop an enterprise procurement application that ties together the inventory and ordering Web services of multiple suppliers and facilitates the procurement process. We first develop the entire application using only the vanilla Web services platform (as described in the first part of the book). After identifying the shortcomings of this implementation based only on the vanilla platform, we add to and expand on the application using the advanced standards and technologies described in the second part of the book.

We conclude this book by summarizing and highlighting some of the key points to remember when developing enterprise Web services and applications.

Summary

Web services represent enormous opportunities and challenges. How organizations negotiate these hurdles will determine the benefits they incur. In this book, we describe the Web services platform—where it is and where it is going—so that developers building applications are cognizant of the fluid nature of the platform and can address enterprise system requirements within the context of a changing platform.

Architect's Note

- Web services are remotely hosted and managed applications whose capabilities can be accessed programmatically by client applications via an addressable URL.
- The core Web services platform, consisting of SOAP, WSDL, and UDDI, provides the means for building distributed applications based on industry standard technologies, interfaces, and protocols.
- The core Web services platform does not provide all of the necessary capabilities on which to build enterprise systems. Additional technologies are being developed and are being standardized that can be layered on top of the core platform and provide support for security and authentication, transactions, mobile and wireless access, quality-of-service, workflows, conversations, systems and service management, as well as interactive applications and Web portals.
- Web services are important and different from other distributed computing environments because they are based on industry standards that are nearly ubiquitous. This allows unprecedented interoperability between applications as well as companies and supports anybody-to-anybody applications.
- The adoption of Web services within enterprises will require fundamental changes to IT organizations that are responsible for deploying and maintaining enterprise systems. In an effort to maintain control over enterprise systems within a Web services environment, IT will take on a more strategic role that is aligned with individual business units and become part of the business decision process.

PART 1

Basic Web Services Standards, Technologies, and Concepts

In this first section of the book, we briefly review the industry standards, technologies and concepts that underlie Web services. These critical technologies support the development of Web services as well as applications that use (or consume) Web services. But, be forewarned that these foundational technologies do not provide everything necessary to build Web services and applications that meet enterprise requirements. We cover these advanced technologies in Section Two of this book.

In this section, we describe the following technologies that together make up the basic Web services platform:

Chapter 2: XML Fundamentals. In this first of three chapters in Part One, we start with a discussion of the fundamentals of the eXtensible Markup Language (XML), the basic technology on which Web services are based. From network protocols up the stack to back-end databases, XML in all its forms has had a commoditizing effect on enterprise computing systems and being both platform and language independent is a natural choice for the level of interoperability required of Web services.

Chapter 3: SOAP and WSDL. Here we describe in detail the two technologies that make up the foundations of Web services: SOAP and WSDL. SOAP (Simple Object Access Protocol) is an XML-based mechanism for exchanging information between applications within a distributed environment. This information exchange mechanism can be used to send messages between applications and, more specifically, can be used to implement remote procedure calls (RPCs). WSDL (Web Services Description Language) is an XML-based language for describing Web services. Through a WSDL description, a client application can determine the location of the remote Web service, the functions it implements, as well as how to access and use each function.

Chapter 4: UDDI. In this chapter, we describe UDDI (Universal Description, Discovery, and Integration), which is a specification for a registry of information for Web services. UDDI defines a means to publish and, more importantly, discover (or search for) information, including WSDL files, about Web services. We also describe the UBR (UDDI Business Registry), which is a global implementation of the UDDI specification.

After reading Section One, you will have a strong understanding of the technologies, standards and concepts underlying Web services. Refer to Section Three for a detailed, step-by-step guide and lots of sample source code to actually develop Web services and client applications.

XML Fundamentals

The suite of technologies grouped under the XML umbrella provides the fundamental building blocks of the Web services architecture. From network protocols through back end databases, XML has had an advantageous effect on enterprise computing systems. Being platform and language independent is a natural choice for building interoperable systems via Web services. Given the importance of XML in enterprise computing, and specifically in Web services, this chapter recaps the fundamentals of XML before embarking on a discussion of more advanced topics such as namespaces and XML Schema.

XML: The Lingua Franca of Web Services

XML is a standard for data mark-up backed by the World Wide Web Consortium, which has been branded "the universal format for structured documents and data on the Web."[1] The entire XML suite of standards, models, and processing technologies have been under development since 1998 with the initial XML specification, and has since been augmented by several additional supporting standards and notes that have brought XML to its current rich state. In fact, though XML is undeniably a richly specified technology, it has retained its simplicity and the entire XML platform can be profiled as follows:[2]

1. From the W3C Web Site at http://www.w3c.org/XML/
2. These (fewer than 10) points are based on the W3C's "XML in 10 Points" available from http://www.w3c.org/XML/1999/XML-in-10-points

- *XML is for Structuring Data*
  Structured data includes things like spreadsheets, address books, configuration parameters, financial transactions, and technical drawings. XML is a set of rules for designing text formats that support the developer in creating structured data. Though it vaguely resembles source code, XML is not a programming language, but it does make it easy for a computer to generate data, read data, and ensure that the data structure is unambiguous. XML avoids common pitfalls in language design. It is extensible, platform-independent, supports internationalization and localization, and is fully Unicode-compliant.
- *XML Resembles HTML*
  Like HTML, XML makes use of tags (words surrounded by angle brackets, "<" and ">") and attributes (of the form *name="value"*). While HTML specifies what each tag and attribute means and often how the text between them will render in a browser, XML uses the tags only to delimit pieces of data and leaves the interpretation of the data completely to the application that reads it.
- *XML is Human Readable, but Humans Shouldn't Read It*
  Programs that produce structured data often store that data on disk, using either a binary or text format. An advantage of a textual format is that it allows people, if necessary, to look at the data without the program that produced it, using tools like text editors. XML files are text files that people shouldn't have to read, but may read as and when the need arises. Care must be taken when manually editing XML since its rules are strict. A forgotten tag or an attribute without quotes makes an XML document unusable. The official XML specification forbids applications from trying to second-guess the creator of a broken XML file; if the file is broken, an application has to stop and report an error.
- *XML is Verbose*
  Since XML is a textual format and uses tags to delimit the data, XML files are nearly always larger than comparable binary formats. That was a conscious decision by the designers of XML. The advantages of a text format are evident, and the disadvantages can usually be compensated at a different level by compression applications. In addition, the transfer of XML across networks can be hastened by communication protocols such as those used in modems protocols and HTTP/1.1, which can compress data on-the-fly, saving bandwidth almost as effectively as a binary format.
- *XML is a Suite of Technologies*
  XML 1.0 is the specification that defines what "tags" and "attributes" are. Beyond that specification, the XML family is a growing set of modules that offer useful services to accomplish important and frequently demanded tasks.
- *XML is Modular*
  XML allows you to define a new document format by combining and reusing other formats. Since two formats developed independently may have elements or attributes

with the same name, care must be taken when combining those formats. To eliminate name confusion when combining formats, XML provides a namespace mechanism that is supported in all XML-based technologies.

- *XML is License-Free, Platform-Independent, and Well-Supported*
  By choosing XML as the basis for Web services, we gain access to a large and growing community of tools and techniques on which to develop value. Basing Web services on XML is similar to basing a database strategy on SQL—you still have to build your own database, programs, and procedures that manipulate it, but there are many tools and commodity components available to help. Furthermore, since XML is license-free, Web services can be built without incurring royalty payments.

While a full discussion of the subject of XML is beyond the scope of this book, before delving deeply into developing Web services it is imperative that at least the basics of XML and XML processing are understood. Although some of the XML detail inherent in developing Web services can be abstracted by toolkits, the increasing popularity of XML at the application level means that any learning at this point will, in addition to accelerating the rate of understanding Web services technology, be generally valuable in day-to-day development. That said, it's time to get acquainted with some fundamental XML concepts.

XML Documents

The purpose of an XML document is to capture structured data, just like an object in an object-oriented programming language. Documents are structured into a number of *elements*, delimited by *tags* which may or may not be nested within other elements.

Anyone familiar with the syntax of HTML will immediately be comfortable with the look and feel of XML, although anyone thinking about coding XML like HTML must be wary—XML is extremely strict in its syntax, where the interpretation of HTML (particularly by browsers) is quite permissive. As we progress through the examples, it is worth remembering the fundamental document syntax:

1. All tags must have corresponding end tags unless they are devoid of subelements, in which case they can be represented as
   `<element-name … attributes … />`.
2. No element can overlap any other element, although nesting within elements is allowed.
3. A document can only have a single root element (which excludes the XML declaration `<?xml … ?>`).
4. Attributes of an element must have unique names within the scope of a single tag.
5. Only element names and attribute name-value pairs may be placed within a tag declaration.

The best way to understand XML is by example, and the XML document shown in Figure 2-1 is typical of the structure of most XML documents, though it is somewhat shorter than most we'll be seeing in the Web services world.

```
<?xml version="1.0" encoding="utf-8"?>
<dvd>
  <title>The Phantom Menace</title>
  <year>2001</year>
</dvd>
```

Figure 2-1 A simple XML document.

Figure 2-1 shows a simple XML document that contains data about a DVD. The document (as all XML documents should) begins with the XML Declaration, delimited by `<?` and `?>`. This declaration provides information for any programs that are going to process the document. In this case it informs any processors that the XML document is encoded according to version 1.0 (at the moment 1.0 is the first and only XML version and the 1.1 effort is underway) and the underlying textual encoding is UTF-8 as opposed to ASCII.

The remainder of the document is where the actual structured data is held. In this case we have a `root` element delimited by the `dvd` tag, which contains two subelements delimited by the `title` and `year` tags. Those subelements contain textual data that we assume relates to the name of the film on the disk and the year of its release (though this is a convention and we could name elements badly, just as we can poorly name variables when programming).

We can take this document one stage further and make it a little more useful for those programs who might want to derive richer information from it. The document shown in Figure 2-2 embellishes that from Figure 2-1 adding in the DVD regional information as an attribute to the root element `region="2"`. We have also added a comment to aid human readability that is delimited by `<!--` and `-->`.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- This is the European release of the DVD -->
<dvd region="2">
  <title>The Phantom Menace</title>
  <year>2001</year>
</dvd>
```

Figure 2-2 A simple XML document with attributes and comments.

The addition of the attribute in Figure 2-2 would, for instance, be of great help to a DVD cataloging system that could use the region attribute to classify disks by their target geographical region.

XML Namespaces

Namespaces in object-oriented programming languages allow developers to name classes unambiguously. Given that different organizations (should) use different namespaces for the software components, even in the cases where two third-party software components contain a class with exactly the same name, the fact that those classes are in different namespaces means that they are easily distinguished from one another.

Unambiguous naming is a requirement that also permeates the XML world. For example, it may be the case that several versions of a document with a root element `dvd` may exist, but the structure of each is different. The way we distinguish the document that we want from a number of available `dvd` documents is by its XML namespace.

Unlike popular programming languages where specific scope resolution operators are used to build namespaces (e.g., `MyPackage.MyClass` in Java and `MyNamespace::MyClass` in C++) the convention in XML is to use a URI (Universal Resource Identifier) as the namespace identifier.

In fact, XML namespaces use URIs by convention only. Strictly speaking, an XML namespace is just a string. The value in using URIs is that they ensure uniqueness that strings cannot.

The URI is the union of the familiar URL and the not-so-familiar URN (Uniform Resource Name) schemes as shown in Figure 2-3 and Figure 2-4.

```
ftp://src.doc.ic.ac.uk
gopher://gopher.dna.affrc.go.jp
http://www.arjuna.com
mailto:some.one@somewhere.com
news:uk.jobs.offered
telnet://foo.bar.com/
```

Figure 2-3 Some familiar URI schemes.

The general scheme for the construction of a URI is `<scheme>:<scheme-specific-part>`. An absolute URI contains the name of the scheme in use followed by a colon (e.g., `news:`), which is followed by a string which is interpreted according to the semantics of that scheme (i.e., `uk.jobs.offered` identifies a particular Usenet newsgroup).

While the URI scheme doesn't mandate the meaning of the `<scheme-specific-part>`, many individual schemes share the same form which most Web users will have experienced with URLs (Uniform Resource Locator) where the syntax consists of a sequence of four parts: `<scheme>://<authority><path>?<query>` (for example, `http://search.sun.com/`

search/suncom/?qt=java). Depending on the scheme in use, not all of these parts are necessary but given those rules any valid URI can be constructed.

> Another good convention to adopt for namespaces is that the URI chosen should have some meaning. For instance, if a document has a namespace which is a HTTP URL, then dereferencing that URL should retrieve the schema which constrains that document.

A URN is intended to be a persistent, location-independent, resource identifier. In typical situations a URN is used where a name is intended to be persistent. The caveat is that once a URN has been affiliated with a particular entity (protocol message, Web service, and so on), it must not be reused to reference another resource. The URNs in Figure 2-4 are typical of the kinds of identifiers we find in Web services applications (taken from OASIS BTP, see Chapter 7):

```
urn:oasis:names:tc:BTP:1.0:core
urn:oasis:names:tc:BTP:1.0:qualifiers
```

Figure 2-4 An example of the URN scheme.

XML namespaces affiliate the elements and attributes of an XML document with namespaces identified by URIs. This process is called qualification and the names of the elements and attributes given a namespace scope are called qualified names, or simply QNames.

Now that we understand we can qualify our documents with a namespace, we can extend the example in Figure 2-2 to include namespace affiliation. Given that it is likely there will be other DVD cataloging systems and those systems will also use elements with names like dvd (which will likely have a different structure and content from our own version), the addition of a namespace into our XML document confers the advantage that it cannot be mixed up with any other similar-looking dvd documents from outside of our namespace. Our newly namespaced document is shown in Figure 2-5.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- This is the European release of the DVD -->
<d:dvd xmlns:d="http://dvd.example.com" region="2">
<d:title>The Phantom Menace</d:title>
<d:year>2001</d:year>
</d:dvd>
```

Figure 2-5 A simple namespaced XML document with attributes and comments.

We have introduced into Figure 2-5 an association between a *prefix* and a *URI* (in this case we've used a URL), using the xmlns attribute from the XML Namespace specification. We then used that prefix throughout the document to associate our elements with that namespace. Any XML processing infrastructure that reads our document does not see the elements as simply their element names but de-references the URI to arrive at the form `{URI}:<local name>` (e.g., `{http://dvd.example.com}:dvd}`) which is unambiguous, unlike the element name alone (i.e., just `dvd`). It is important to remember that the syntax `{prefix}:<local name>` is not understood by XML processing programs, it is a convention used when describing qualified elements.

Although any element can contain a namespace declaration, the style convention in XML is to declare all namespaces that a document uses in its root element. Although this can make the opening tag of the root element quite large, it does improve overall document readability since we do not then pepper the document with namespace declarations.

Explicit and Default Namespaces

XML permits two distinct kinds of namespace declarations. The first of these as we have seen is the *explicit* form, whereby a prefix is given a namespace association (e.g., `xmlns:d="http://dvd.example.com"`), and then elements and attributes which belong to that namespace are explicitly adorned with the chosen prefix. The second of these is the default namespace declared as `xmlns=<uri>` that provides a default namespace affiliation which applies to any elements without a prefix.

The default namespace can be used to improve the readability of an XML document. In documents where a particular explicit namespace is predominantly used (like the WSDL or SOAP documents in Chapter 3), declaring a default namespace alleviates the need to pepper the document with the same prefix all over. Using this strategy, only those elements outside of the default namespace will need to be prefixed, which can make documents significantly easier to understand.

We present a modified version of the XML from Figure 2-5 in Figure 2-6, where the default namespace declaration implicitly scopes all following elements within the `http://dvd.example.com` namespace, like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- This is the European release of the DVD -->
<dvd xmlns="http://dvd.example.com" region="2">
<title>The Phantom Menace</title>
<year>2001</year>
</dvd>
```

Figure 2-6 Using default namespaces.

Adding a namespace affiliation to an XML document is analogous to placing a Java class into a specific package. Where the Java equivalent of in Figure 2-2 (which has no namespace affiliation) might have been referenced by a declaration such as `DVD myDVD`, the equivalent type of reference for the document in Figure 2-5 or Figure 2-6 would be `com.example.dvd.DVD myDVD`, which when reduced to Java terms is clearly unambiguous since only the owner of the `dvd.example.com` domain should be using that namespace (and by inference should be the only party using that namespace to name XML documents).

Inheriting Namespaces

Once a default or explicit namespace has been declared, it is "in scope" for all child elements of the element where it was declared. The default namespace is therefore propagated to all child elements implicitly unless they have their own explicit namespace.

This arrangement is common in WSDL files (Chapter 3) where the WSDL namespace is the default namespace for an interface, but where the binding elements use their own explicit namespace.

The rule of thumb for choosing a default or explicit namespace is that if you can't see at a glance yourself which namespace an element belongs to, then no one else will be able to and, therefore, explicit namespaces should be used. If, however, it is obvious which namespace an element belongs to and there are lots of such elements in the same namespace, then readability may be improved with the addition of a default namespace.

And Not Inheriting Namespaces

Of course, a child may not necessarily want to inherit the default namespace of its parent and may wish to set it to something else or remove the default namespace entirely. This is not a problem with explicit namespaces because the child element can just be prefixed with a different explicit namespace than its parent, as shown in Figure 2-7, where the genre element has a different namespace affiliation than the rest of the document (which uses the default namespace).

```
<?xml version="1.0" encoding="utf-8"?>
<!-- This is the European release of the DVD -->
<dvd xmlns="http://dvd.example.com" region="2">
    <title>The Phantom Menace</title>
    <year>2001</year>
    <g:genre xmlns:g="http://film-genre.example.com">
        sci-fi
    </g:genre>
</dvd>
```

Figure 2-7 Mixing explicit and default namespaces within a document.

It is important to realize that any children of the `genre` element in Figure 2-7 that use the default namespace will be using the default namespace of the `dvd` element since the genre element only declares an explicit namespace for its scope. Similarly, with default namespaces, any element is at liberty to define a namespace for itself and any of its children irrespective of the namespace affiliations of any of its parent elements. This is shown below in Figure 2-8:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- This is the European release of the DVD -->
<dvd xmlns="http://dvd.example.com" region="2">
    <title>The Phantom Menace</title>
    <year>2001</year>
    <genre xmlns ="http://film-genre.example.com">
        sci-fi
    </genre>
</dvd>
```

Figure 2-8 Mixing default namespaces within a document.

The `genre` element from Figure 2-8 declares that the default namespace for itself and its children (if any) are, by default, in the namespace `http://film-genre.example.com`. This differs from the example shown in Figure 2-7 since in the absence of any explicit namespace, children of the genre element belong to the `http://film-genre.example.com` and not to the `http://dvd.example.com` namespace as the outer elements do.

Of course it may be the case that an element does not require a default namespace and that the parent default namespace is inappropriate. In such cases, we can remove any default namespace completely, by setting it to the empty string `xmlns=""`.

For default namespaces, remember that the scoping rules are based on the familiar concept of "most local" where the declaration nearest to the use has the highest precedence.

Attributes and Namespaces

So far all of our attention has been focused on the interplay between namespaces and elements. However, it is equally valid for attributes to be qualified with namespaces through the same prefix syntax. When namespace-qualifying attributes have a default namespace, different rules apply compared to elements. Attributes are not affiliated with any default namespace, so if an attribute is to be namespace qualified, then it must be done so explicitly since any attribute without a prefix will not be considered namespace qualified—even if declared in the scope of a valid default namespace.

The convention in XML is to associate elements with namespaces, but to leave attributes unqualified since they reside within elements with qualified names.

At this point we now understand both basic XML document structure and some more advanced features like namespaces. These both set the scene for higher-level XML-based technologies (including Web services) which we shall continue by looking at XML Schema.

XML Schema

With the exception of the basic XML syntax, XML Schema is without a doubt the single most important technology in the XML family. In the Web services world, XML Schema is *the* key technology for enabling interoperation.

XML Schema is a W3C recommendation[3] that provides a type system for XML-based computing systems. XML Schema is an XML-based language that provides a platform-independent system for describing types and interrelations between those types. Another aspect of XML Schema is to provide structuring for XML documents.

Document Type Definitions (or DTDs) were the precursor to XML Schema, and are a text- (not XML-) based format designed to convey information about the structure of a document. Unlike XML Schema, DTDs do not concern themselves with type systems, but simply constrain documents based on their structure. Furthermore, since the DTD language is not XML-based, many of the XML-friendly tools that we use are incapable of processing DTDs. Because of these reasons, and the fact that no recent Web services protocols have used DTDs, we can consider DTDs as a deprecated technology in the Web services arena. Instead, XML Schema has become the dominant metadata language for Web services (and indeed for most other application areas by this time).

In fact, the analogy between XML technologies and object-orientation is clear if we compare XML documents to objects and XML Schema types to classes. XML documents that conform to a schema are known as instance documents, in the same way that objects of a particular class are known as instances. Thus we can conceptually match XML Schema schemas with classes and XML documents with objects, as shown in Figure 2-9.

3. See http://www.w3.org/XML/Schema#dev for links to the XML Schema specifications.

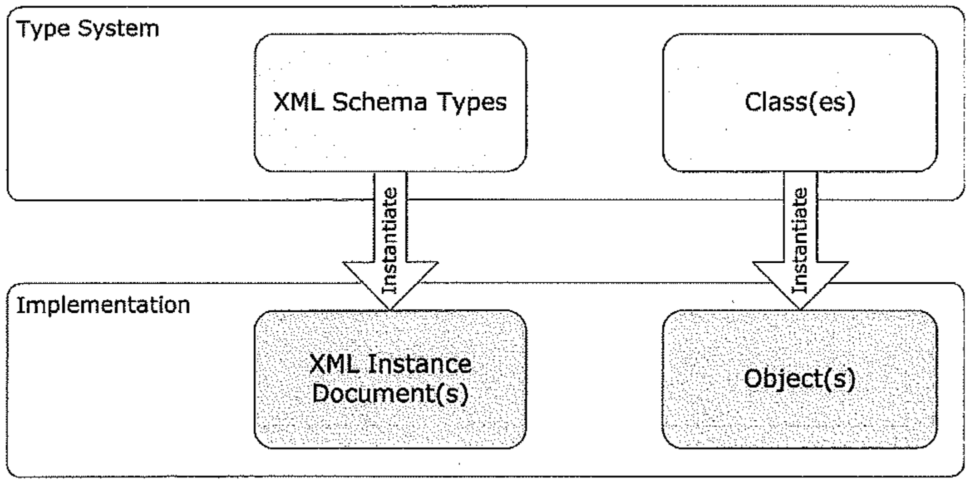


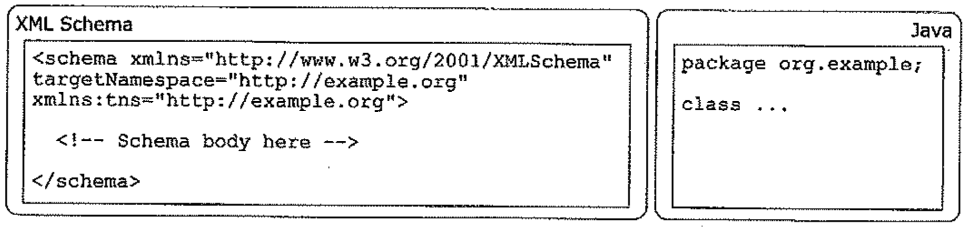
Figure 2-9 Comparing XML to object-oriented model.

The conceptual relationship between an object model and XML Schema is straightforward to comprehend. Where object-based systems classes and their interrelationships provide the blueprint for the creation and manipulation of objects, in the XML arena it is the type model expressed in XML Schema schemas that constrain documents that confirm to those schemas.

Like object-oriented programming languages, XML Schema provides a number of built-in types and allows these to be extended in a variety of ways to build abstractions appropriate for particular problem domains. Each XML Schema type is represented as the set of (textual) values that instances of that type can take. For instance the `boolean` type is allowed to take values of only `true` and `false`, while the `short` type is allowed to take any value from `-32768` to `32767` inclusively. In fact, XML Schema provides 44 different built-in types specified in the http://www.w3.org/2001/XMLSchema namespace. Additionally, XML Schema allows users to develop their own types, extending and manipulating types to create content models is the very heart of XML Schema.

XML Schema and Namespaces

As we have seen, the built-in types from XML Schema are qualified with the namespace http://www.w3.org/2001/XMLSchema. We must not use this namespace when we develop our own types, in the same way that we would not develop types under the `java.lang` package in Java or `System` namespace in .Net. However, like adding package or namespace affiliations in object-oriented programming, affiliating a type with a namespace in XML Schema is straightforward. Adding a `targetNamespace` declaration to an XML Schema to affiliate it with a namespace is analogous to adding a `package` declaration to a Java class, as shown in Figure 2-10.



```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://example.org"
xmlns:tns="http://example.org">

  <!-- Schema body here -->

</schema>
```

```
package org.example;

class ...
```

Figure 2-10 Adding namespace affiliation to an XML schema.

The skeletal schema shown in Figure 2-10 outlines the basic principle on which all XML Schema operate: the schema element delimits the namespace (like the keyword `package` delimits the package scope for a single Java source file) and the `targetNamespace` gives the namespace a name (like the period-separated string that follows the `package` keyword).

Don't be confused by the number of namespaces that exist in Figure 2-10. There are in fact only two of them and they play three distinct roles. The default namespace is the XML Schema namespace because the elements that we use in this document, such as the root element `<schema>`, are from the XML Schema namespace. The `targetNamespace` namespace is used to declare the namespace which the types that will be declared in this schema will be affiliated with. Finally, the explicit namespace `tns` (an abbreviation of Target NameSpace) will be used to allow types and elements within this schema to reference one another and, hence, it shares the same URI as the `targetNamespace` element.

A First Schema

Now that we understand the basics of XML Schema construction, we can write a simple schema with which we can constrain a document. This simple schema example does not explore any of the type-system features of XML Schema, but instead concentrates on constraining a simple document as a first step. Drawing on our experience with DVD documents earlier in this chapter we will create a schema that can validate a given DVD document. Let's recap the document that we want to constrain in Figure 2-11:

```
<?xml version="1.0" encoding="utf-8"?>
<dvd xmlns="http://dvd.example.com" region="2">
<title>The Phantom Menace</title>
<year>2001</year>
</dvd>
```

Figure 2-11 An XML document containing DVD information.

If we analyze the document in Figure 2-11, we see that it contains an element called dvd, which itself contains two elements, title and year, which are all qualified with the namespace http://dvd.example.com. From this, we immediately know that the targetNamespace is http://dvd.example.com. We also know that the schema requires two nested elements and a globally scoped element, and so we can construct the schema, as shown in Figure 2-12:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://dvd.example.com"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified" >
  <element name="dvd">
    <complexType>
      <sequence>
        <element name="title" type="string"/>
        <element name="year" type="positiveInteger"/>
      </sequence>
      <attribute name="region" type="positiveInteger"/>
    </complexType>
  </element>
</schema>
```

Figure 2-12 A first DVD schema.

Since the elements in the document in Figure 2-11 have a namespace that matches the targetNamespace of the schema in Figure 2-12, we can assume that the document is a valid instance of the schema.

The schema dictates that the instance document must have an opening element with the name dvd from the line <element name="dvd"> at the opening line of the schema body.

The conventional style for XML Schema documents is to declare the opening element with elementFormDefault= "qualified" and attributeFormDefault="unqualified" to ensure that elements in instance documents should be namespace qualified by default, while any attributes should lack any namespace qualification.

The schema then goes on to declare that there should be a sequence of two nested elements within that first dvd element, called title and year, respectively. Specifying this is done with four elements. The first of these is the complexType element which indicates that the parent dvd element consists of other elements nested within it. Inside the complexType element we see a sequence element. A sequence element places the constraint on any conformant document that elements nested within must follow the same sequence as the schema.

In this case, since the elements nested within the sequence are the `title` element followed by the `year` element, conformant documents must also specify `title` before `year`. The `title` element must contain information in string form because its type attribute is set to the `string` type from the XML Schema namespace. Similarly, the year element specifies that its information must be encoded as an XML Schema `positiveInteger` type.

The final aspect of this schema is to describe that the outer-most `dvd` element requires an attribute to hold region information. This constraint is applied with the `<attribute>` element which mandates an attribute called `region` whose value must be of type `positiveInteger`.

While we can now begin to create simple schemas to constrain simple documents, scaling this approach to large schemas and large documents is usually impractical and undesirable. Instead we need to look beyond the document—which after all is only the serialized, readable form of XML—to the real power of XML Schema: its type system.

Implementing XML Schema Types

The real beauty of XML Schema is that once a document has been validated against a schema, it becomes more than just a set of elements and tags—it becomes a set of types and instances. The elements contained within a document are processed and the type and instance information from them is exposed to the consuming software agent. After validation, the information contained in an XML Document is called a post schema-validation Infoset, or usually an Infoset. Infosets make it possible to reflect over the logical contents of a document, just like in some object-oriented programming languages, and so the power of XML Schema as a platform-independent type system is revealed. To demonstrate, let's start to build some types and see how the (logical) type system works with the (physical) document.

Creating Simple Types via Restriction

XML Schema provides a total of 44 simple types with which to build content models. However, unlike simple types in most programming languages, in XML Schema these types can be used as base types for the creation of specialized subtypes. There is a key difference though when we define a subtype of a simple type in XML Schema, in that we do not change the structure of the type (as we would do when we inherit from a base class in Java), but instead change the subset of values that the subtype can handle. For instance we might specify a subtype of the simple type `string` that can only be used to hold a value that represents a postcode. Similarly we might restrict the date type to valid dates within a particular century.

We create a subtype of a simple type in XML Schema using the `restriction` element. Within this element, we specify the name of the simple type whose set of permissible values we will be restricting (known as the base type) and how exactly the restriction will be applied. Restrictions are then specified by constraining *facets* of the base simple type, where the set of available facets in XML Schema is shown in Figure 2-13.[4]

4. Information from Part 2 of the XML Schema Specification at http://www.w3.org/TR/xmlschema-2/

Facet Element	Description
`length`	Specifies the number of characters in a string-based type, the number of octets in a binary-based type, or the number of items in a list-based type.
`minLength`	For string datatypes, minLength is measured in units of characters. For hexBinary and base64Binary and datatypes, minLength is measured in octets of binary data. For list-based datatypes, minLength is measured in number of list items.
`maxLength`	For string datatypes, maxLength is measured in units of characters. For hexBinary and base64Binary datatypes, maxLength is measured in octets of binary data. For list-based datatypes, maxLength is measured in number of list items.
`pattern`	Constrains the value to any value matching a specified regular expression.
`enumeration`	Specifies a fixed value that the type must match.
`whiteSpace`	Sets rules for the normalization of white space in types.
`maxInclusive`	Constrains a type's value space to values with a specific inclusive upper bound.
`maxExclusive`	Constrains a type's value space to values with a specific exclusive upper bound.
`minInclusive`	Constrains a type's value space to values with a specific inclusive lower bound.
`minExclusive`	Constrains a type's value space to values with a specific exclusive lower bound.
`fractionDigits`	For decimal types, specifies the maximum number of decimal digits to the right of the decimal point.
`totalDigits`	For number types, specifies the maximum number of digits.

Figure 2-13 XML schema facets.

Each of the facets shown in Figure 2-13 allows us to constrain simple types in a different way. For example, to create a simple type that can be used to validate a British postal code, we would constrain a string type using the `pattern` facet with a (complicated) regular expression as shown in Figure 2-14.

```
<simpleType name="PostcodeType">
  <restriction base="string">
    <xs:pattern value="(GIR 0AA)|((([A-Z][0-9][0-9]?)|(([A-
Z][A-HJ-Y][0-9][0-9]?)|(([A-Z][0-9][A-Z])|([A-Z][A-HJ-Y][0-
9]?[A-Z])))) [0-9][A-Z]{2})"/>
  </restriction>
</simpleType>
```

Figure 2-14 The pattern facet.

The pattern specified in Figure 2-14 allows only values that match the British postal code standard, such as SW1A 1AA (the Prime Minister's residence in 10 Downing Street) or W1A 1AE (the American Embassy in London). Formally, these rules are defined by the British Post Office[5] as:

1. The first part of the code before the space character (known as the outward code) can be 2, 3 or 4 alpha-numeric characters followed by a space and the second part of the code (the inward code), which is 3 characters long and is always 1 digit followed by 2 alpha-characters. Permitted combinations according to the `PostcodeType` type are: AN NAA, ANN NAA, AAN NAA, AANN NAA, ANA NAA, AANA NAA, (where A=alpha character and N=numeric character).
2. The letters I and Z are not used in the second alpha position (except GIR 0AA which is an historical anomaly in the British postal system).
3. The second half of the code never uses the letters C, I, K, M, O, and V.

Any divergence from this form will mean that the element is not a valid `PostcodeType` instance.

Similarly, we might want to create an enumeration where only specific values are allowed within a type, such as those for currencies. An example of this is shown in Figure 2-15, where the XML Schema `string` type is restricted to allow only certain values that represent a number of world currencies:

```
<xs:simpleType name="CurrencyType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="GBP"/>
    <xs:enumeration value="AUD"/>
    <xs:enumeration value="USD"/>
    <xs:enumeration value="CAD"/>
    <xs:enumeration value="EUR"/>
    <xs:enumeration value="YEN"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 2-15 The pattern facet.

5. http://www.govtalk.gov.uk/gdsc/schemaHtml/BS7666-v1-xsd-PostCodeType.htm

The CurrencyType declared in Figure 2-15 would validate elements such as <my-currency>GBP</my-currency>, but would not validate <your-currency>DM</your-currency> since the string DM is not part of this simpleType restriction (nor for that matter are Deutsch Marks any longer legal tender).

Continuing in a monetary theme, we can create StockPriceType type where we specify that the number of digits after the decimal point is at the most 2. In Figure 2-16 we restrict the XML Schema decimal type such that the maximum number of digits after the decimal point in a stock price is 2. This type can then be used to validate elements that have the form <msft>25.52</msft> and <sunw>3.7</sunw>:

```
<xs:simpleType name="StockPriceType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 2-16 The fractionDigits facet.

To specify sizes of allowed values, we use the length, maxLength and minLength facets. For instance, a sensible precaution to take when creating computer passwords is to mandate a minimum length for security and a maximum length for ease of use (and thus indirectly for security). In XML Schema, we can use maxLength, and minLength facets to create a PasswordType as shown in Figure 2-17:

```
<xs:simpleType name="PasswordType">
  <xs:restriction base="xs:string">
    <xs:minLength value="6"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 2-17 maxLength and minLength facets.

When applied to an element in a document, the PasswordType in Figure 2-17 allows values like <password>katherine</password>, but does not allow for values such as <password>carol</password> based on the number of characters contained in the element. Of course if a particularly overbearing system administration policy was put into place, we could end up having passwords of a long, fixed length using the length facet instead of minLength and maxLength.

In much the same way that we set the maximum and minimum number of characters with the maxLength, minLength and length facets, we can also specify the maximum and minimum values. Specifying a range of values is achieved with the maxInclusive, minInclusive, minExclusive and maxExclusive facets. For instance, we may wish to define the range of seconds in a minute for timing purposes. A simpleType called SecondsType is shown in Figure 2-18, where the int type from XML Schema is constrained to accept the values from 0 (inclusive) to 59 (60 exclusive):

```
<xs:simpleType name="SecondsType">
  <xs:restriction base="xs:int">
    <xs:minInclusive value="0"/>
    <xs:maxExclusive value="60"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 2-18 minInclusive and maxExclusive facets.

Similarly we might want to define the years in a particular century, as we see in Figure 2-19, where the years that are part of the 20th century are captured as being positive integers (which have the range from 1 upward) from 1901 (1900 exclusive) through to 2000 (inclusive):

```
<xs:simpleType name="TwentiethCenturyType">
  <xs:restriction base="xs:positiveInteger">
    <xs:minExclusive value="1900"/>
    <xs:maxInclusive value="2000"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 2-19 minExclusive and maxInclusive facets.

The totalDigits facet puts an upper limit on the number of digits that a number-based type can contain. For example a year number, for around the next 8000 years, contains a total of four digits. Thus, we can create a simple year type using the totalDigits facet to constrain the number of digits to four, as shown in Figure 2-20 where the positiveInteger type from XML Schema is restricted to those positive integers which have at most 4 digits:

```
<xs:simpleType name="YearType">
  <xs:restriction base="xs:positiveInteger">
    <xs:totalDigits value="4"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 2-20 The totalDigit facet.

The final facet for restricting the value space of simple types is `whiteSpace`. This facet allows a simple type implementer to specify how any white spaces (tabs, spaces, carriage returns, and so on) are handled when they appear inside elements. There are three options for the `whiteSpace` facet which are: `preserve` (the XML processor will not remove any white space characters), `replace` (the XML processor will replace all white space with spaces), and `collapse` (same as replace, with all preceding and trailing white space removed).

Often the `whiteSpace` facet is applied along with other facets to deal with extraneous white space. For instance if we add a `whiteSpace` facet to the `YearType` from Figure 2-20, the XML processor that processes instances of this type can deal with any unimportant white space in it. This is shown in Figure 2-21, where the `whiteSpace` facet is set to `collapse`, which effectively rids the value of any unwanted white space after it has been processed:

```
<xs:simpleType name="YearType">
  <xs:restriction base="xs:positiveInteger">
    <xs:totalDigits value="4"/>
    <xs:whiteSpace value="collapse"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 2-21 The whiteSpace facet.

So, if the XML processor receives an element of type `YearType such as:`

```
<moon-landing>
    1969
```

`</moon-landing>`, the `whiteSpace collapse` facet will effectively reduce it to `<moon-landing>1969</moon-landing>`.

The built-in simple type `NormalizedString` will automatically strip line feeds, carriage returns or tabs from any white spaced text.

Simple Type: List and Union

Though restriction is one means of creating new simple types, it is not the only way. XML Schema supports two additional mechanisms for creating new simple types: `union` and `list`.

Both `union` and `list` are aggregation mechanisms, and so there is no type hierarchy. Therefore we cannot "cast" between base type and union or list type as we can with types derived through `restriction`.

The `list` mechanism is the simpler of the two to understand. In short, simple types created via the `list` mechanism are a white space-delimited list of values from the base type. For example, we can create a list of instances of `YearType` from Figure 2-20 to create the YearsType as shown in Figure 2-22:

```
<xs:simpleType name="YearType">
  <xs:restriction base="xs:positiveInteger">
    <xs:whiteSpace value="collapse"/>
    <xs:totalDigits value="4"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="YearsType">
  <xs:list itemType="YearType"/>
</xs:simpleType>
```

Figure 2-22 Creating new simple types with list.

The `YearsType` type defined in Figure 2-22 can then be used to validate instances of the `YearsType` such as the `years` element in Figure 2-23.

```
<WWII> 1939 1940 1941 1942 1943 1944 1945 1946</WWII>
```

Figure 2-23 An instance of the YearsType type.

The `union` mechanism is slightly more subtle than the `list`. It allows the aggregation of the value spaces of two types to be combined into the value space of a new single simple type. For instance, imagine we have two simple types that represent fruits and vegetables, respectively, as shown in Figure 2-24:

```
 <xs:simpleType name="FruitType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="ORANGE"/>
    <xs:enumeration value="APPLE"/>
    <xs:enumeration value="BANANA"/>
    <xs:enumeration value="KIWI"/>
  </xs:restriction>
</xs:simpleType>

  <xs:simpleType name="VegetableType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="POTATO"/>
    <xs:enumeration value="CABBAGE"/>
    <xs:enumeration value="TURNIP"/>
    <xs:enumeration value="LEEK"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 2-24 FruitType and VegetableType simple types.

We can use the `FruitType` and `VegetableType` types in Figure 2-24 to create a `FruitAndVegetableType` via a `union` as shown here in Figure 2-25:

```
<xs:simpleType name="FruitAndVegetableType">
  <xs:union memberTypes="FruitType VegetableType"/>
</xs:simpleType>
```

Figure 2-25 Creating a new simple type via a union.

The resulting `FruitAndVegetableType` type can be used to validate elements such as `<organically-grown>BANANA</organically-grown>` and `<menu-item>POTATO</menu-item>` because both `BANANA` and `POTATO` are valid values for the `FruitAndVegetableType` type.

Simple Type Support in Programming Languages

The XML Schema support for simple user-defined types that allow custom value and lexical spaces is a powerful aspect of the technology. However, since most programming languages do not support this feature, typically programmers have had to produce properties/accessors/mutators that constrain the value space by manually checking values and throwing exceptions where constraints have been invalidated. For example, take the Java equivalent of the `YearType` type (from Figure 2-20) shown in Figure 2-26:

```
public class Year
{
  public int getValue()
  {
    return _value;
  }

  public void  setValue(int value)
                         throws InvalidValueException
  {
    if(value >=  1000 &&  value <= 9999)
    {
      _value = value;
    }
    else
    {
      // Invalid year
      throw new InvalidValueException();
    }

  }

  private  int  _value;
}
```

Figure 2-26 Value and Lexical handling with Java's primitive types.

The `Year` class in Figure 2-26 is somewhat lengthier than the equivalent XML Schema simple type since it has to handle the value space imperatively rather than declaratively. To deal with the lexical space of year instances, we need to manually check the possible values and report back to the user when an invalid value is encountered as exemplified in the `Year.setValue(int)` method.

Writing these kinds of classes by hand is long-winded and prone to error. Of course we could provide tool support to deal with these issues (like the `xsd.exe` tool from the .Net platform toolkit), but if we are dealing with schematized XML documents, it happens that we don't necessarily need to. Consider the diagram in Figure 2-27 of a typical XML-enabled software agent (which could be a standalone application, a database, or more likely a Web service) that communicates with its environment through schematized XML documents.

The ability to define custom value/lexical spaces that fit our precise needs means that it is possible to delegate constraint checking of values in an XML document to the XML processor. Once the XML processor has produced an Infoset for the program to consume, the XML document that the Infoset was created from must have passed validation by its schema, and so the value and lexical constraints placed on the documents must be satisfied. Knowing this, the devel-

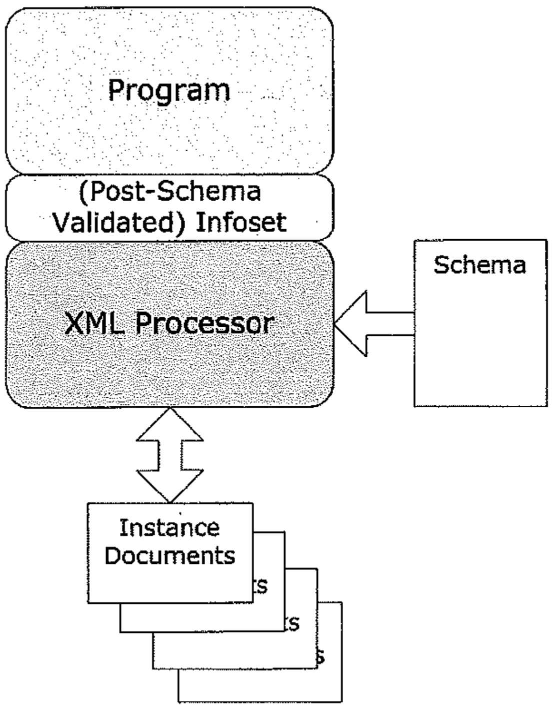


Figure 2-27 Delegating Value/Lexical space error handling to the XML processor.

oper of the consuming program no longer has to write lengthy constraint checking code since this would be a replication of work that the XML processor already undertakes. Thus using schemas can remove some of the burden of manually checking values in our code, though it is not a substitute for failing to program defensively!

Complex Types

As well as creating specialized versions of the XML Schema simple types, we can also create new complex types by aggregating existing types into a structure. XML Schema supports three means of aggregating types with three different complex type compositors: `sequence`, `choice`, and `all` whose semantics are outlined in Figure 2-28.

Compositor	Description
`sequence`	Specifies that the contents of the complex type must appear as an ordered list.
`choice`	Allows a choice of any of the contents of the complex type.
`all`	Specifies that the contents of the complex type appear as a unordered list.

Figure 2-28 complexType compositors.

While the semantics of the compositors vary, the syntax of each is quite similar. To use any of the compositors, we simply declare a new complex type with a compositor as its child element, as shown here in Figure 2-29:

```
<xs:complexType name="AddressType">
  <xs:sequence>
    <xs:element name="number" type="xs:string"/>
    <xs:element name="street" type="xs:string"/>
    <xs:element name="city" type="xs:string"/>
    <xs:element name="state" type="xs:string"/>
    <xs:element name="post-code" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="business-address" type="xs:boolean"/>
</xs:complexType>
```

Figure 2-29 Declaring a new complexType using the sequence compositor.

In Figure 2-29 we create a new complexType called AddressType by aggregating five elements of type string which represent a mailing address, and a single attribute of type boolean which is used to indicate whether this address is business or residential.

In the scope of a sequence compositor, each contained element must appear exactly once by default. If more flexibility is needed, then we can add the minOccurs and maxOccurs attributes to each contained element. The minOccurs attribute is set to a value greater than or equal to 0 which then specifies the minimum number of occurrences for its element within the compositor. The maxOccurs attribute specifies the maximum number of elements that should appear in the compositor from 1 to the special value unbounded (which is logically an infinite number of times).

With the AddressType in Figure 2-29, we can now validate elements such as the address element in Figure 2-30:

```
<address>
  <number>221b</number>
  <street>Baker Street</street>
  <city>London</city>
  <state>N/A</state>
  <post-code>NW1 6XE</post-code>
</address>
```

Figure 2-30 A valid instance of the AddressType type.

The all compositor is similar to the sequence compositor except that ordering constraint is relaxed. Therefore while the elements contained within an all compositor must be present, the order in which they appear is unimportant from the point of view of the XML processor.

The minOccurs and maxOccurs attributes do not make sense in the scope of an all compositor since (for example) it is impossible to specify that an instance document should contain all the instances of a maxOccurs="unbounded" element! Instead, omitting these attributes gives us the default semantics of exactly one element per compositor. The only exception here is that minOccurs="0" can be used to specify optional elements.

An example of the all compositor is shown in Figure 2-31, where the PurchaseOrderType type is presented. The PurchaseOrderType uses the all compositor to create an aggregate structure containing mandatory `order-number` and `item` elements, and an optional `description` element (specified by the `minOccurs="0"` attribute):

```
<xs:complexType name="PurchaseOrderType">
  <xs:all>
    <xs:element name="order-number"
      type="xs:positiveInteger"/>
    <xs:element name="item" type="xs:string"/>
    <xs:element name="description" type="xs:string"
      minOccurs="0"/>
  </xs:all>
</xs:complexType>
```

Figure 2-31 Using the all compositor.

The `PurchaseOrderType` type from Figure 2-31 can be used to validate the instances shown in Figure 2-32, where we see instances both where the description element is missing and where it is present:

```
<purchase-order>
  <order-number>1002</order-number>
  <item>11025-32098</item>
  <description>Personal MP3 Player</description>
</purchase-order>

<purchase-order>
  <item>44045-23112</item>
  <order-number>5290</order-number>
</purchase-order>
```

Figure 2-32 Valid PurchaseOrderType instances.

Using the `choice` compositor, we can force the contents of part of a document to be one of a number of possible options. For example, in Figure 2-33 we see the `UserIdentifier-Type`, which allows a user to supply either a login identifier or Microsoft Passport-style single-signon credentials to log in to a system (this type of arrangement is typical in e-commerce sites).[6]

6. Note that this is a hypothetical example that has been deliberately shortened for clarity, and the types used are not representative of the actual Passport API.

```
<xs:complexType name="UserIdentifierType">
  <xs:choice>
    <xs:element name="login-id" type="xs:string"/>
    <xs:element name="passport" type="xs:anyURI"/>
  </xs:choice>
</xs:complexType>
```

Figure 2-33 Using the choice compositor.

The `UserIdentifierType` can be used to validate elements that contain either a `login-id`, or a `passport` element, but not both. Therefore both the elements shown in Figure 2-34 can be validated against the `UserIdentifierType`:

```
<logon>
  <login-id>chewbacca@wookie.org</login-id>
</logon>

<logon>
  <passport>
    http://passport.example.org/uid/2235:112e:77fa:9699:aad1
  </passport>
</logon>
```

Figure 2-34 Valid UserIdentifierType elements.

The `minOccurs` and `maxOccurs` attributes can be used within `choice` compositor. They allow us to expand the basic exclusive OR operation that `choice` provides, to support selection based on quantity as well as content, as exemplified in Figure 2-35:

```
<xs:complexType name="DrinksMenuType">
  <xs:choice>
    <xs:element name="beer" type="b:BeerType" minOccurs="0"
      maxOccurs="2"/>
    <xs:element name="wine" type="w:WineType" minOccurs="0"
      maxOccurs="1"/>
  </xs:choice>
</xs:complexType>
```

Figure 2-35 Choosing elements based on cardinality.

Using the `DrinksMenuType` type, we can specify using the `minOccurs` and `maxOccurs` attributes that our choice can be either two beers or one drink of wine, as shown in Figure 2-36.

```
<!-- Either two beers… -->
<drinks>
  <b:beer type="bitter"/>
  <b:beer type="lager"/>
</drinks>

<!-- … Or a single drink of wine -->
<drinks>
  <w:wine country="France" grape="Pinot Noir" year="1998"/>
</drinks>
```

Figure 2-36 Instance documents constrained by choice.

Equally, we could select based on quantity of a single item. For example we could envision a `choice` where beer can be sold in four, six and twelve packs by simply setting the `minOccurs` and `maxOccurs` attributes to 4, 6 and 12, respectively, as shown in Figure 2-37:

```
<xs:complexType name="DrinksMenuType">
  <xs:choice>
    <xs:element name="beer" type="xs:string" minOccurs="4"
      maxOccurs="4"/>
    <xs:element name="beer" type="xs:string" minOccurs="6"
      maxOccurs="6"/>
    <xs:element name="beer" type="xs:string" minOccurs="12"
      maxOccurs="12"/>
  </xs:choice>
</xs:complexType>
```

Figure 2-37 Choice based on cardinality.

With `choice`, we have drawn to a close our discussion on compositors. We have seen how we can aggregate existing types into new types in a variety of ways (`sequence`, `choice`, `all`) and some of the variations on those themes (like choice-by-cardinality). However, we can also create new types not only by aggregating existing types, but by aggregating existing types and textual content. For instance, we might wish to mix textual information and structured data to create a letter[7] as shown in Figure 2-38.

7. This example adapted from the W3 Schools example at: http://www.w3schools.com/schema/schema_complex_mixed.asp

```
<letter>
Dear Professor <name>Einstein</name>,
Your shipment (order: <orderid>1032</orderid> )
will be shipped on <shipdate>2003-06-14</shipdate>.
</letter>
```

Figure 2-38 Mixed textual and element content.

In order to mix elements and text, we must create a type that allows such mixtures (and by default types do not). Thus we create a schema such as that shown here in Figure 2-39:

```
<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="orderid" type="xs:positiveInteger"/>
      <xs:element name="shipdate" type="xs:date"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 2-39 Schema supporting mixed textual and element content.

The way that we support mixed textual and elemental content is to create a `complexType` with `mixed` content. Thus when the `mixed` attribute is set to `true` (in its absence the default is `false`), the resulting type can mix elements and text as shown in the letter example in Figure 2-38.

The `any` Element

By default, all complex types that we create have closed content models. This means that only the elements that are specified when the type is declared can appear in instances. While this certainly encourages strong typing, it can also be a problem. How do we handle elements within a document that we cannot predict ahead of time? Indeed many of the Web services protocols that we will encounter in later chapters have this requirement, where the content model of schemas for particular protocols has to be extended on a per application basis (in fact, we discuss how WS-Transaction extends WS-Coordination in this way in Chapter 7). Fortunately this kind of extensibility is supported in XML Schema through the `any` element, which allows us to develop an open content model for a type through the use of wildcards.

Using `any` within a complex type means that any element can appear at that location, so that it becomes a placeholder for future content that we cannot predict while building the type. For attributes, there is the `anyAttribute` which defines placeholders for future attribute extensions.

Of course, we might not want to allow completely arbitrary content to be embedded, and so `any` can be constrained in a number of ways, but don't worry, it will still be generic even after the constraints. The first constraint that we can place on `any` is how the contents that are substituted will be treated by the XML processor. The `processContents` attribute has a number of options that can be chosen to set the level of validation of elements specified by an any element. These are:

- `strict`—This is the default value in the absence of any `processContents` attribute. The XML processor must have access to the schema for the namespaces of the substituted elements and fully validate those elements against that schema.
- `lax`—This is similar to strict, with the exception that if no schema can be located for substituted elements, then the XML parser simply checks for well-formed XML.
- `skip`—This is the least taxing validation method, which instructs the XML processor not to validate any elements from the specified namespaces.

The namespaces against which the contents may be validated are specified by a second optional attribute for the `any` element called `namespace`. This attribute specifies the namespace of the elements that it is valid to substitute for an any element within a document, and has a number of possible settings:

- `##any`—This is the default setting for the namespace attribute which implies that elements from any namespace are allowed to exist in the placeholder specified by the `any` element.
- ##`other`—Specifying this value for the namespace attribute allows elements from any namespace except the namespace of the parent element (i.e., not the `targetNamespace` of the parent).
- `##local`—The substituted elements must come from no namespace.
- `##targetNamespace`—Only elements from the namespace of the parent element can be contained.

Finally we are allowed to combine some of the above options to make the available namespaces more configurable. That is, we are allowed to specify a space-separated list of valid namespace URIs (instead of `##any` and `##other`), plus optionally `##targetNamespace` and `##local`. Thus we can restrict the namespaces for which it is valid to substitute any element to a list of (one or more) specific namespaces if necessary.

An example of how the `any` element is used is presented in Figure 2-40.

```
<xsd:complexType name="Notification">
  <xsd:sequence>
    <xsd:element name="TargetProtocolService"
      type="wsu:PortReferenceType"/>
    <xsd:element name="SourceProtocolService"
      type="wsu:PortReferenceType" />
    <xsd:any namespace="##other" processContents="lax"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:anyAttribute namespace="##other"
    processContents="lax"/>
</xsd:complexType>
```

Figure 2-40 WS-Transaction messages are extensible via any and anyAttribute.

Figure 2-40 shows the `Notification` type from the WS-Transaction protocol schema. A `Notification` in WS-Transaction is a message that is transmitted between actors in the protocol. However, since WS-Transaction is designed to allow different back end transaction systems to operate on the Internet, the messages it exchanges have to be extensible enough to express the semantics of each back end system. This, of course, calls for an open content model to allow third parties to extend the protocol to suit their own systems.

The protocol supports wildcard elements and attributes via the `xsd:any` and `xsd:anyAttribute` elements. In both cases, the wildcard element namespaces must come from any namespace other than the WS-Transaction namespace as its `namespace` attribute is set to `##other`. This is exemplified in Figure 2-41 where we see a SOAP message (see Chapter 3 for a full explanation of SOAP) from one vendor's WS-Transaction implementation (see Chapter 7 for details on Web services transactions) using the wildcard elements to propagate information pertinent to their implementation.[8]

Although the `DialogIdentifier` element from the SOAP message in Figure 2-41 wasn't specified by the schema, it is still a valid message because it matches the constraints of the `<xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>` element from the schema. It matches the `##other` constraint since it comes from the namespace http://schemas.arjuna.com/ws/2003/01/wsarjtx which is valid since the WS-Transaction namespace is http://schemas.xmlsoap.org/ws/2002/08/wstx. Since the schema maintains that processing of these elements is lax, it means that the XML processor that receives this message will validate the well-formed XML of the `DialogIdentifier` element. Thus the message conforms to the schema even though the originators of the schema had no idea about the organization that ultimately created the conformant message, let alone the message itself.

8. This SOAP message is from Arjuna Technologies' XTS 1.0 implementation of the WS-Transaction protocol.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <wstx:OnePhaseCommit xmlns:wstx="http://schemas.xmlsoap.org/ws/
2002/08/wstx">
      <wstx:TargetProtocolService xmlns:wstx="http://
schemas.xmlsoap.org/ws/2002/08/wstx">
        <wsu:Address xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/
07/utility">
http://localhost:5555/jboss-net/services/TwoPCParticipantMSG
        </wsu:Address>
      </wstx:TargetProtocolService>
      <wstx:SourceProtocolService xmlns:wstx="http://
schemas.xmlsoap.org/ws/2002/08/wstx">
        <wsu:Address xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/
07/utility">
http://localhost/jboss-net/services/TwoPCCoordinator
        </wsu:Address>
      </wstx:SourceProtocolService>
      <wsarjtx:DialogIdentifier xmlns:wsarjtx=
        "http://schemas.arjuna.com/ws/2003/01/wsarjtx">
        123456
      </wsarjtx:DialogIdentifier>
    </wstx:OnePhaseCommit>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 2-41 Using Wildcard element to extend a WS-Transaction message.

In addition to the `any` and `anyAttribute` elements, XML Schema also provides two special types called `anyType` and `anySimpleType` which can be used instead of a specific named type where we need our schemas to be more generic.

The `anyType` type is the most generic of the two being substitutable for any type in the whole XML Schema type system, including user-derived types. The `anySimpleType` is more constrained and supports only types that are from the set of forty-four XML Schema simple types or types derived from them.

These special types provide the same kind of generality when creating type-based content models as the `any` element provides for document structure. It is not unusual to see attributes like `type="xs:anyType"` or `type="xs:anySimpleType"` in element declarations where the type of such elements is expected to be determined by the application that consumes the schema, and not by the schema developer.

Inheritance

While the ability to constrain instance documents is essential for interoperability, harnessing the type system exploits the real power of XML Schema. The inheritance features in XML Schema allow us to create type hierarchies that capture the data models of the underlying software systems that XML is designed to support.

In fact, we have already seen one form of inheritance when we used the `restriction` feature to create new simple types with differently constrained value and lexical spaces. However, XML Schema also supports a mechanism called `extension` that allows us to augment (rather than constrain) the capabilities of an existing type. Using this facility we can begin to create hierarchies of complex types just as we can in object-oriented programming languages.

When using complex type extension, we have two options for creating subtypes. We can create subtypes that contain only simple content (text and attributes only), or subtypes that contain complex content (other elements as well as text and attributes).

An example of extending a complex type with additional simple content is shown in Figure 2-42:

```
<xs:complexType name="MonitorType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="flatscreen" type="xs:boolean"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Figure 2-42 Complex Type `extension with simpleContent.`

The `MonitorType` complex type in Figure 2-42 uses the `simpleContent` element to add a single attribute to its content, which is defined as being the `string` built-in type. The base type of the `MonitorType (string)` is specified by the `base` attribute in the `extension` element. The additional simple content is specified as the only child of this `extension` element. The new subtype we have defined can now be used to validate elements such as `<monitor flatscreen="true">HP P4831D</monitor>`.

Figure 2-43 shows an example of how we can use the `extension` mechanism to create subtypes with additional elements using the `complexContent` construct.

```
<xs:complexType name="PersonType">
  <xs:sequence>
    <xs:element name="forename" type="xs:string"/>
    <xs:element name="surname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="FootballerType">
  <xs:complexContent>
    <xs:extension base="PersonType">
      <xs:sequence>
        <xs:element name="team" type="xs:string"/>
        <xs:element name="goals" type="xs:int"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Figure 2-43 Complex Type `extension with complexContent`.

The `PersonType` type in Figure 2-43 can be used to validate instances such as that shown here in Figure 2-44:

```
<person>
  <forename>Alan</forename>
  <surname>Turing</surname>
</person>
```

Figure 2-44 An Instance of the `PersonType` Type.

The `FootballerType` in Figure 2-43 has `complexContent`, allowing the elements and attributes to appear within the body of the type. It capitalizes on that fact by adding the `team` and `goals` elements to extend on the base `PersonType` to allow the validation of such elements as shown in Figure 2-45:

```
<footballer>
  <forename>Alan</forename>
  <surname>Shearer</surname>
  <team>Newcastle United</team>
  <goals>145</goals>
</footballer>
```

Figure 2-45 A `FootballerType` Type Instance.

As we see in Figure 2-45, instances of the `FootballerType` type have a similar structure to instances of the `PersonType` type, because the `FootballerType` subtype inherits the forename and surname elements from the `PersonType`, but adds the elements team and goals.

From this example, we can see that it is possible to use the `extension` mechanism to build type hierarchies in XML Schema, just as we can in object-oriented programming languages. However, to be able to exploit such hierarchies (e.g. to "cast" between types) we need to use another XML Schema mechanism: substitution groups.

Substitution Groups

Substitution groups are a feature that allows us to declare that an element can be substituted for other elements in an instance document. We achieve this by assigning an element to a special group—a substitution group—that is substitutable for the element at the *head* of that group, effectively creating an equivalence relation between document elements of the same type (or subtype).

Elements in a substitution group must have the same type as the head element, or a type that has been derived from the head element's type.

While this isn't exactly like polymorphic behavior in object-oriented programming languages since the base-type/derived type relationship isn't implicit, this feature is immensely useful for creating extensible schemas with open content models.

To illustrate this point, consider the schema shown in Figure 2-46. This schema demonstrates how to use substitution groups to deal with element-level substitutions—a kind of polymorphic behavior for instance documents. The substitution group consists of the elements `cast-member` and `crew-member` declaring themselves to be substitutable for a person element through the `substitutionGroup="person"` attribute declaration. Note that this is a valid substation group because both `cast-member` and `crew-member` are types derived from the `PersonType` type.

The definition of the `cast-and-crew` element references the `person` element from within a `sequence`, setting the `maxOccurs` attributes to allow any number of `person` elements to exist within an instance. However, since person is an abstract element (and thus cannot appear as an element in its own right), this schema actually supports the substitution of `person` elements for any other element declared to be in the same substitution group. Therefore, this schema will validate instance documents such as that shown in Figure 2-47.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="PersonType">
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"
        minOccurs="0"/>
      <xs:element name="surname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="CastMemberType">
    <xs:complexContent>
      <xs:extension base="PersonType">
        <xs:sequence>
          <xs:element name="character" type="xs:string"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="CrewMemberType">
    <xs:complexContent>
      <xs:extension base="PersonType">
        <xs:sequence>
          <xs:element name="function" type="xs:string"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <!-- Declare substitution group and head element -->
  <xs:element name="person" type="PersonType"
    abstract="true"/>
  <xs:element name="cast-member" type="CastMemberType"
    substitutionGroup="person"/>
  <xs:element name="crew-member" type="CrewMemberType"
    substitutionGroup="person"/>
  <!-- Now define the actual document -->
  <xs:element name="cast-and-crew">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="person" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2-46 Using substitution groups.

```
<?xml version="1.0" encoding="UTF-8"?>
<cast-and-crew>
  <crew-member>
    <firstname>Lucas</firstname>
    <surname>George</surname>
    <function>director</function>
  </crew-member>
  <cast-member>
    <firstname>Ewan</firstname>
    <surname>McGregor</surname>
    <character>Obi Wan Kenobi</character>
  </cast-member>
</cast-and-crew>
```

Figure 2-47 Supporting polymorphic behavior with substitution groups.

The instance document in Figure 2-47 shows how types from the `person` substitution group can be used in places where the original schema has specified a `PersonType` element. In this case since both `cast-member` and `crew-member` are part of the `person` substitution group, the document is valid.

Like the `any` and `anyAttribute` elements, substitution groups are a useful mechanism for creating schema types which are extensible. Again like the `any` and `anyAttribute` elements, substitution groups are widely found in various Web services standards. WSDL (see Chapter 3) makes extensive use of substitution groups to allow other protocols (such as BPEL, see Chapter 6) to extend its basic features to more complex problem domains.

Global and Local Type Declarations

Just like classes in object-oriented programming, we need to create instances of XML Schema types in order to do real work like moving XML encoded messages between Web services. In this section, we examine two means for creating instances of types: using global types and declaring local types.

We have already seen examples of both of global (schema-scoped) and local (element-scoped) type declarations throughout the previous sections. A global type definition occurs where we embed a type directly as a child of the `<schema>` element of a schema. Conversely, a local type is declared as the child an `<element>` element, which is a direct child of the `<schema>` element. This is exemplified in Figure 2-48.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <!-- A Global Type -->
  <xs:complexType name="CardType">
    <xs:sequence>
      <xs:element name="card-type">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="Visa"/>
            <xs:enumeration value="MasterCard"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="expiry">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:pattern value="[0-9]{2}-[0-9]{2}"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="number">
        <xs:simpleType name="CardNumberType">
          <xs:restriction base="xs:string">
            <xs:pattern
              value="[0-9]{4} [0-9]{4} [0-9]{4} [0-9]{4}"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="holder" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <!-- A local type -->
  <xs:element name="debit-card">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="CardType">
          <xs:attribute name="issue"
            type="xs:positiveInteger"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <!-- Another local type -->
  <xs:element name="wallet">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="credit-card" type="CardType"
          minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="debit-card" ref="debit-card"
          minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2-48 Global and Local type declarations.

The distinction between the two is important. Global types such as `CardType` in Figure 2-48 are globally visible and so are available within the namespace in which they are declared and in other namespaces, can be extended and generally behave as we would expect classes to behave in an object-oriented programming language. Instances of global types are created by constructing an element whose `type` attribute refers to that particular global type's name. This is shown in Figure 2-48 where we see this element:

```
<xs:element name="credit-card" type="CardType"
  minOccurs="0" maxOccurs="unbounded"/>
```

that defines that an instance of the `CardType` type can be present any number of times in a wallet element.

On the other hand, local types are declared inline with an element (like `debit-card` and `wallet` in Figure 2-48). While the element itself is visible to other elements and types, its implementing type is not and therefore is not extendable by other types—in fact, the implementing type doesn't even have a name so that it can be referred to.

When we declare local types, they can subsequently be referred to only by their enclosing element name and their content cannot be extended. In programming terms, this is similar to a component whose API is known, but whose type is anonymous and internal structure is a black box. This is shown in Figure 2-48 where the `wallet` (itself a local type) is defined as containing any number of instances of the `debit-card` local type via the `ref` attribute, like this:

```
<xs:element name="debit-card" ref="debit-card"
            minOccurs="0" maxOccurs="unbounded"/>.
```

Instances of local types are specified by the ref attribute, e.g., `<xs:element name="credit-card" ref="debit-card" … />` Instances of global types are specified by type attribute, e.g., `<xs:element name="credit-card" type="CardType" … >`

Whether to declare types globally or locally depends on our intended use for those types. If we intend for those types to form part of a type hierarchy, then they should be declared globally so they can be extended at will. If, however, we intend for a type to only support instances within XML documents, then it should be declared locally.

A good rule of thumb for developing content models is to type hierarchies with global types, but to create local type declarations at the leaf nodes of those hierarchies. Thus within the hierarchy we have the full flexibility supplied by global types, yet the "interface" presented to users of that hierarchy is a collection of element declarations against which XML documents can be validated.

Managing Schemas

While most of the schemas we have seen in this chapter have been short, it is possible for schemas that serve particularly complicated problem domains to become long and difficult to manage. XML Schema helps to solve this problem by providing the `include` mechanism that allows us to partition a single logical schema (i.e., the set of types from a single `targetNamespace`) across a number of physical schema documents. For instance, we could choose to create type hierarchies in one physical document and create the document layout in another physical document for ease of management. These two separate physical documents can then be made into a single logical schema by including the type hierarchy document in the document structure schema, as shown in Figure 2-49 and Figure 2-50.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
targetNamespace="http://wallet.example.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:complexType name="CardType">
    <xs:sequence>
      <xs:element name="card-type">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="Visa"/>
            <xs:enumeration value="MasterCard"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="expiry">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:pattern value="[0-9]{2}-[0-9]{2}"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="number">
        <xs:simpleType name="CardNumberType">
          <xs:restriction base="xs:string">
            <xs:pattern
              value="[0-9]{4} [0-9]{4} [0-9]{4} [0-9]{4}"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="holder" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 2-49 The Type hierarchy part of the Wallet schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
targetNamespace="http://wallet.example.com" xmlns:tns="http://
wallet.example.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:include schemaLocation="CreditCard.xsd"/>
  <xs:element name="debit-card">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="tns:CardType">
          <xs:attribute name="issue"
            type="xs:positiveInteger"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="wallet">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="credit-card" type="tns:CardType"
          minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="tns:debit-card" ref="debit-card"
          minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2-50 The Document-Structure part of the Wallet schema.

The schema shown in Figure 2-49 effectively becomes the container for all of the types that might be used in the XML documents that conform to the schema (which at the moment is only a single type, `CardType`). The schema in Figure 2-50 uses the include mechanism to create a single logical schema containing itself and the included schema from Figure 2-49. This gives access to all of the types defined in the included schema, allowing the `wallet` to be constructed in the same way as it was when the two schemas were physically one (in Figure 2-48), with the advantage that because the individual schemas are smaller, maintaining them is easier.

While the `include` mechanism is fine for partitioning a single schema across multiple physical schema documents, it is limited to schema documents which share the same `targetNamespace`. It is easy to see the limitation of this mechanism if we imagine for a moment that the definition of the `CardType` had not been developed by the same in-house team that created the wallet, but had instead been created by an outside consortium of credit card companies. In this case the `targetNamespace` will be different from that of the wallet schema and so `include` will not work. Instead, we use the `import` mechanism, which allows us to combine types and elements from different namespaces into a single schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
targetNamespace="http://card.example.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:complexType name="CardType">
  <!-- Card implementation omitted for brevity -->
  </xs:complexType>
</xs:schema>
```

Figure 2-51 The Credit Card schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
targetNamespace="http://wallet.example.com" xmlns:tns="http://
wallet.example.com" xmlns:cc="http://card.example.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:import namespace="http://card.example.org"
    schemaLocation="CreditCard.xsd"/>
  <xs:element name="debit-card">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="cc:CardType">
          <xs:attribute name="issue"
            type="xs:positiveInteger"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="wallet">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="credit-card" type="cc:CardType"
          minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="debit-card" ref="tns:debit-card"
          minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2-52 The Wallet schema.

The schema in Figure 2-51 declares a single type (`CardType`) in the namespace http://card.example.com. The schema containing the `CardType` type is then exposed to the schema shown in Figure 2-52 via the `import` mechanism, which involves specifying both the namespace that is being imported and the location of the schema which is attributed with that targetNamespace.

The imported namespace is given a prefix (so that it can be referenced within the wallet schema) via the `xmlns:cc` attribute in the root element of the wallet schema document. Now the components of the credit card schema (including `CardType`) are accessible to the wallet schema by referencing its qualified name (`QName`) via the prefix `cc`.

Once we have imported a schema, we can freely reference its contents. In the wallet schema, we use the contents of the credit card schema to create a new type of card (`debit-card`) by extending the credit card schema's `CardType`. We also create a `wallet` element that declares instances of both the global `CardType` and instances of the local `debit-card` type. As we have seen, the `import` declaration works just like an `import` declaration in the Java programming language or using a declaration in C#, which simply exposes the types from a foreign namespace to the current namespace.

Schemas and Instance Documents

Until this point we have largely focused on either XML documents or constructing portable type systems with XML Schema. However, it is only when these two aspects of XML intersect that we actually have a usable technology for moving structured data between systems. That is, we need to be able to communicate the abstract notions defined in schemas via concrete XML documents and on receipt of an XML document, be able to translate it back into some form suitable for processing within the receiving system—which is generally an Infoset or native object model, not a mass of angle brackets and text. The relationship between types, elements and instance documents is captured in Figure 2-53.

Schema-aware XML processors (like Apache's Xerces and the .Net System.XML classes) use an instance document's namespace to match against the corresponding namespace of a schema. However, the XML Schema specification doesn't mandate how the XML processor should locate that schema in the first place. Typically, an XML processor will be programmatically or administratively configured with the locations of any required schemas before undertaking any processing. However, this can be restrictive in that the schemas of all possible instance documents must be known ahead of time if they are to be validated by the XML processor.

While the XML Schema specification doesn't provide a means of mandating the location of a schema, it does provide a means of *hinting* at its location by placing and `xsi:schemaLocation` attribute into the instance document, as shown in Figure 2-54.

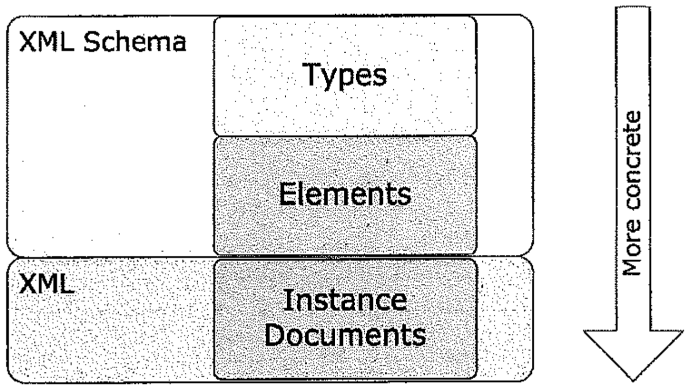


Figure 2-53 Relationship between Types, Elements and Documents.

```
<ptr:printer xmlns:p="http://printer.example.org"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://printer.example.org
file:/home/local/root/schemas/printer.xsd">
  <!-- rest of schema omitted for brevity -->
```

Figure 2-54 Using the `xsi:schemaLocation` attribute to locate a schema.

The `xsi:schemaLocation` attribute specifies a set of space-delimited namespace-location pairs indicating the location of schemas for particular namespaces. Upon finding the `xsi:schemaLocation` attribute, the XML processor *may* (since it is only a hint) try to obtain the specified schema from the suggested location. Of course, the processor may not try to obtain this information from the `xsi:schemaLocation` attribute, especially if it already has the necessary document-schema mappings through other means.

XML Schema Best Practices

We've now seen a great deal of XML Schema, and over time we have built up a set of informal best practices based on the notion of defining important global types and their interrelations first and document structure later. However, it is useful to condense these details down to their barest bones for quick reference:

1. Always use `elementFormDefault="qualified"` and `attributeFormDefault="unqualified"` to ensure that elements are namespaced by default and attributes are not.
2. Declare all types globally; declare elements (apart from the document root) locally.

3. Use types to express content models, use elements to dictate the structure of documents.
4. Use the XML Schema features that most closely match your object model. Do not map the object model onto a different model in Schema just because it makes writing schemas easier.

These best practices are intended as guidelines. Over time you will develop your own practices that more accurately match the kinds of solutions you are working on. However, the fact remains that no matter what style we ultimately develop for Web services projects, we still need to use XML to move data around systems.

Processing XML

To round off this discussion on XML technology, it is worth taking a brief look at some of the means of processing the XML documents and schemas that we have so far examined to see how we traverse from the XML level to the application level. There are a number of standard, cross-platform tools available that perform much of the hard work involved in processing XML. In this section we concentrate on three of the most prevalent XML processing technologies: SAX, DOM, and XSLT.

The examples we have chosen to illustrate the technologies are necessarily simple. In each example we simply harvest the character information from a simple DVD document as shown Figure 2-55.

With each XML processing tool, we take the XML shown in Figure 2-55 and present it as an XML fragment such as:

```
<d:character xmlns:d="http://dvd.example.org">
        Qui Gon Jin
    </d:character>
    <d:character xmlns:d="http://dvd.example.org">
        Queen Amidala
    </d:character>
    <d:character xmlns:d="http://dvd.example.org">
        Obi Wan Kenobi
    </d:character>
    <d:character xmlns:d="http://dvd.example.org">
        Anakin Skywalker
    </d:character>
    <d:character xmlns:d="http://dvd.example.org">
        Senator Palpatine
    </d:character>
```

which could then be used as the basis for other processing.

```
<?xml version="1.0" encoding="utf-8"?>
<d:dvd xmlns:d="http://dvd.example.org" region="2">
  <d:title>The Phantom Menace</d:title>
  <d:year>2001</d:year>
  <d:language>
    <d:audio>English</d:audio>
    <d:subtitle>Danish</d:subtitle>
    <d:subtitle>Norwegian</d:subtitle>
    <d:subtitle>Swedish</d:subtitle>
    <d:subtitle>English</d:subtitle>
  </d:language>
  <d:actors>
    <d:actor firstname="Liam" surname="Neeson">
      <d:character>Qui Gon Jin</d:character>
    </d:actor>
    <d:actor firstname="Natalie" surname="Portman">
      <d:character>Queen Amidala</d:character>
    </d:actor>
    <d:actor firstname="Ewan" surname="McGregor">
      <d:character>Obi Wan Kenobi</d:character>
    </d:actor>
    <d:actor firstname="Jake" surname="Lloyd">
      <d:character>Anakin Skywalker</d:character>
    </d:actor>
    <d:actor firstname="Ian" surname="McDiarmid">
      <d:character>Senator Palpatine</d:character>
    </d:actor>
  </d:actors>
  <d:directors>
    <d:director firstname="George" surname="Lucas">
      <d:favorite-film>
                The Empire Strikes Back
            </d:favorite-film>
    </d:director>
  </d:directors>
  <d:barcode>5039036007375</d:barcode>
  <d:price currency="sterling">19.99</d:price>
</d:dvd>
```

Figure 2-55 A complex XML document.

SAX: Simple API for XML

The SAX model is based on the notion of a fast, forward-only and low memory footprint method of processing XML documents. To achieve these goals, the SAX parsers read through an XML document firing events whenever they encounter certain interesting parts of the document (in addition to having the ability to check documents against schemas). As it happens, those parts which the SAX parser seeks are wide-ranging and consist of everything from finding the beginning of a document (and its end) through to catching the occurrence of every open and close tag, and any textual data in between those tags.

To work with SAX, the application code must register for events that it is specifically interested in. For example, we might be particularly interested in extracting the details of a DVD from one of our `dvd` documents in order to store those details in some database. To achieve that, we would need to register the features of the document that we are interested in with the SAX parser. Then when the SAX parser parses the document, it will then inform us each time one of those features is encountered and our application code can use those signals to build up its own object model.

To use a SAX implementation within an application, as developers we must write code that subscribes to SAX events and pieces together a set of objects (or other structured data) from the events that the parser generates. The burden on the developer is to create a document handler capable of listening for the salient events being issued by the SAX parser and write a suitable object model to encapsulate data exposed by the SAX events.

If the object model developed to deal with the SAX events is lightweight, the SAX-oriented aspects of an application can be made lightning fast since SAX itself is also lightweight. The downside is, of course, that the document handler might be non-trivial to develop, especially for complex documents.

To illustrate, let's write some code to harvest the character information from a `dvd` document using the Java program shown in Figure 2-56. This is an undeniably long piece of code for essentially stripping out a few elements. Its length is due to the fact that the SAX parser only deals with creating events and not with the structured data associated with those events. In fact, the overwhelming size of this document is pared down to handling the various events that the SAX parser will issue as it parses a `dvd` document.

The `startElement` and `endElement` methods are called by the SAX parser when an element is entered or exited, respectively, and we use that event to determine whether we have found a character element. If we have found a `character` element, we simply set the `_characterFound` flag to `true`, whereas if we have not found a character element or if we are leaving an element altogether, then the flag is set to `false`.

The `characters(...)` method is called by the SAX parser whenever character data is encountered. If we are within a character element, i.e., the `_characterFound` flag is `true`, then we simply store the character data. All other character data is ignored. The main method simply sets up the parser and parses the document before pretty-printing the resulting characters to standard output.

On a positive note, the SAX approach offers good performance since we tailor the object model exactly to our needs (in this case it's just a linked list of characters). On a less positive note, SAX can be a complex tool to implement with due to the large number of possible events that we might have to write handlers for.

```
import java.io.*;
import java.util.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class SAXExample extends DefaultHandler
{
    // Constants
    private static final String _MY_DVD_NAMESPACE_URI =
                                        "http://dvd.example.com";
    private static final String _CHARACTER_ELEMENT_NAME =
                                                 "character";

    // Flag to remember if we are dealing with character
    // data while parsing
    private boolean   _characterFound = false;

    // The data we're looking for in the document
    private LinkedList _characters    = new LinkedList();

    /**
     * The method called when the start of a new element is
     * found.
     */
    public void startElement(String namespaceURI,
                             String localName,
                             String qualifiedName,
                             Attributes attributes)
                            throws SAXException
    {
        // If the element is called "character" and is in the
    // namespace "http://dvd.example.com" we've found one.

        _characterFound =
         namespaceURI.toLowerCase()
         .equals(_MY_DVD_NAMESPACE_URI) &&
         localName.toLowerCase()
         .equals(_CHARACTER_ELEMENT_NAME);
    }

    /**
     * The method called when the end of an element is found.
     */
    public void endElement(String namespaceURI,
                           String localName,
                           String qualifiedName)
                          throws SAXException
    {
        _characterFound = false;
    }
```

Figure 2-56 Creating a SAX-based application in Java.

```
/**
 * The method called when character data is found.
 */
public void characters(char[] ch, int start, int length)
                      throws SAXException
{
    if(_characterFound)
    {
        _characters.add(new String(ch, start, length));
    }
}

/**
 * A convenience method to pretty-print the characters
 * found.
 */
public StringWriter outputCharacters()
{
    StringWriter sw = new StringWriter();
    for(int i = 0; i < _characters.size(); i++)
    {
        sw.write("<character xmlns=\"" +
                  _MY_DVD_NAMESPACE_URI  + "\">");
        sw.write((String)_characters.get(i));
        sw.write("</character>\n");
    }
    return sw;
}

/**
 * The starting point of the application.
 */
public static void main(String[] args) throws Exception
{
    // Check to see that we have a single URI argument
    if(args.length != 1)
    {
        return;
    }

    SAXExample saxExample = new SAXExample();
    XMLReader parser = null;

    // Create parser
    try
    {
         parser = XMLReaderFactory.createXMLReader(
                   "org.apache.xerces.parsers.SAXParser");
```

Figure 2-56 Creating a SAX-based application in Java (continued).

```
                // Tell the parser which object will handle
                // SAX parsing events
                parser.setContentHandler(saxExample);
            }
            catch (Exception e)
            {
                System.err.println("Unable to create Xerces SAX
                                      parser - check classpath");
            }

            try
            {
                // The URL that sources the DVD goes here
                //(i.e. perform a GET on some remote Web server).
                parser.parse(args[0]);

                // Dump the character information to screen.
                System.out.println(
                    saxExample.outputCharacters().toString());
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
}
```

Figure 2-56 Creating a SAX-based application in Java (continued).

DOM: Document Object Model

DOM goes one step further than SAX and actually provides a simple tree-based object model on top of the basic XML processing and schema validation capabilities, usually built on top of an underlying SAX parser. When programming with a DOM parser, our application code interacts with an in-memory tree representation of the XML document. As such, DOM parsers are usually more heavyweight processors than their SAX equivalents since irrespective of the complexity or length of the XML document being processed, the same type of tree-based hierarchy is built.

Though this might not be the best data structure for any given application, the fact that DOM provides a simple object model "out of the box" is enticing and because of its simplicity, DOM has gained popularity. Indeed, we would generally only use SAX in preference to DOM where we have stringent performance requirements that rule out creating copies of documents in-memory, or where the tree-like mode of DOM is entirely unsuitable for the actual characteristics of the intended object model. Of course, it is possible to layer our own object model on top of that provided by DOM, thus providing both a natural fit for our application and leveraging DOM's ease-of-use. However, when using DOM as the basis for our own object models, we should be aware that we are consuming memory twice over—once for our own objects and once for DOM.

Like SAX, the DOM API is well planned and straightforward to understand. To show some of features of DOM, we shall revisit the same DVD example that we previously tackled with SAX and illustrate the differences between the two approaches via the C# example shown in Figure 2-57.

```
using System;
using System.Xml;

public class DOMExample
{
    private string getXMLDocument(string url)
    {
        // Grab the dvd document from its source
        System.Net.WebClient wc = new System.Net.WebClient();
        byte[] webData = wc.DownloadData(url);

        // Get the downloaded data into a form suitable for
        // XML processing
        char[] charData = new char[webData.Length];
        for(int i = 0; i < charData.Length; i++)
        {
            charData[i] = (char)webData[i];
        }

        string xmlStr = new String(charData);

        // Clean up the document (first "<" and last ">" and
        // everything in between)
        int start = xmlStr.IndexOf("<", 0,
                                   xmlStr.Length - 1);
        int length = xmlStr.LastIndexOf(">") - start + 1;

        // Return only the XML document parts
        return xmlStr.Substring(start, length);
    }

    public static void Main(string[] args)
    {
    // Check to see that we have a single URI argument
    if(args.Length != 1)
    {
        return;
    }
```

Figure 2-57 Creating a DOM-Based application in C#.

```
    string url = args[0];
        DOMExample domExample = new DOMExample();

        System.Xml.XmlDocument xmlDoc =
                                new System.Xml.XmlDocument();
        xmlDoc.LoadXml(domExample.getXMLDocument(url));

        // Search DOM tree for a set of elements with
        // particular name and namespace
        XmlNodeList xmlNodeList =
                    xmlDoc.GetElementsByTagName("character",
                                   "http://dvd.example.com");

        for(int i = 0; i < xmlNodeList.Count; i++)
        {
            // Dump the contents of the elements we've found
            // to standard output.
            Console.WriteLine(xmlNodeList.Item(i).OuterXml);
        }
    }
}
```

Figure 2-57 Creating a DOM-Based application in C# (continued).

The simple DOM-based application presented in Figure 2-57 is somewhat shorter than the previous SAX-based application. This simplicity does not stem from a different programming language or platform since (even platform zealots must agree) there is little difference between Java and .Net for simple XML processing. The gain in simplicity stems from the DOM processing model which automatically builds a data-structure to hold the contents of the XML document, and provides a simple API for searching and manipulating structure.

In fact the overwhelming majority of this application is spent checking that we have a clean XML document to deal with before we put it into our XML processing components. Since we chose to deal with the results of our remote call as an array of bytes returned via HTTP, we had to convert those bytes to characters and those characters to string, and then ensure that string did not contain any extraneous characters (such as the HTTP header information).

Once we are satisfied that we have our document in a clean form, we then submit it to the .Net DOM infrastructure. Internally, the infrastructure builds the DOM tree for us, and then to extract the character data it is simply a matter of searching for the element name (`character`) in the correct namespace (`http://dvd.example.com`). This search results in a list of possible answers, which we then dump to standard output.

While this is a suitable approach for a trivial example, this DOM-based method might not scale well in production environments. We are paying the price for the ease of use we have enjoyed in terms of memory and processing overhead. So while working with DOM is ultimately easier than SAX programmatically, it is always helpful to think about performance metrics and worth bearing in mind that SAX may be a better choice for some problems.

Extensible Stylesheet Transformation (XSLT) and XML Path Language (XPATH)

XSL is the acronym the W3C has assigned to the "Extensible Stylesheet Language." It consists of a language for transforming XML documents (XSLT) and an expression language used to access or reference parts of an XML document (XPath). It also refers to a formatting language called XML Formatting Objects (or XML-FO), but when most people talk about XSL what they are really talking about is XSLT and XPath. It is this subset of XSL technology that we investigate in this section.

The idea behind XSLT is to provide a declarative, rule-based XML scripting language that can be used to specify transformations on documents—that is, to turn a document from one form into another based on some transformation rules. The benefit of this approach is that we can apply commodity XML processing tools to the processing of XML itself—a recursive and inventive way of bootstrapping XML with XML. XPath supports XSLT by allowing parts of documents undergoing transformations to be referenced. Interestingly enough, XPath is *not* an XML-based syntax since its originators saw the value in being able to embed XPath expressions inside URIs and other non-XML identifiers. The canonical use of XPath is shown in Figure 2-58 where a trivial example of XSLT (with similarly simple XPath expressions) is presented.

XPath 1.0 has become perhaps the most important of the XSL technologies in the Web services arena and is now heavily used in other technologies like BPEL (see Chapter 6).

The stylesheet presented in Figure 2-58 is straightforward—mainly because we haven't tried to do anything too ambitious—and it is far shorter than either the SAX or even DOM versions of the code. The opening line of the document introduces some namespaces and defines what the result of the transformation will be without the prefix `d`. The subsequent six declarations tell the XSLT processor to do nothing with each of the elements that are named. For example, when the XSLT engine encounters a `year` element as a child of a `dvd` element, it triggers the execution of the matching `template`, which performs no processing. The end result of this "empty" `template` is that no output appears for the given element.

The template matching the element expressed in XPath as `/d:dvd/d:actors/d:actor/d:character` (i.e., the `character` element under the `actor` element, contained within the `actors` and `dvd` elements) does something slightly more ambitious. We create a new element in our output that has the same name as the current element we are examining (`character`), which is achieved by assigning the result of the XSLT `name()` function to the value held by the `name` attribute. We also give the newly created element a namespace (which, again, we borrow from the element currently under scrutiny) by referencing its namespace declaration (`namespace-uri()`) and assigning that value to the default namespace attribute for this element in our output.

```
<xsl:stylesheet version="1.0"
    xmlns:d="http://dvd.example.com"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    exclude-result-prefixes="d">

    <!-- We are not creating a document, so remove the
           document declaration -->
    <xsl:output method="xml" omit-xml-declaration="yes"/>

    <!-- Do nothing with these elements -->
    <xsl:template match="d:dvd/d:title"/>
    <xsl:template match="d:dvd/d:year"/>
    <xsl:template match="d:dvd/d:language"/>
    <xsl:template match="d:dvd/d:directors"/>
    <xsl:template match="d:dvd/d:barcode"/>
    <xsl:template match="d:dvd/d:price"/>
    <!-- Extract the value held by and character elements
         encountered -->
    <xsl:template match=
        "d:dvd/d:actors/d:actor/d:character">
        <xsl:element name="{name()}"
            namespace="{namespace-uri()}">
            <xsl:value-of select="."/>
        </xsl:element>
    </xsl:template>
</xsl:stylesheet>
```

Figure 2-58 A simple XSLT stylesheet.

The `value-of` element is then used in combination with the `select="."` attribute to select the value held within the current matching element—where the axis "." is defined as "current context" in XPath. The net result of applying this template is to place the character information for each character encountered into the output from the XSLT engine and wrap that character data inside an appropriately namespaced XML element.

Although the example here has been necessarily trivial (since our goals were similarly trivial), XSLT is a powerful means of transforming XML documents. However, even this basic knowledge of what XSLT (and XPath) is and how it can be applied to XML documents will stand us in good stead as we finally venture out into the Web services world.

Summary

XML is the fundamental technology that underpins everything else in Web services. Of paramount importance to the XML suite of technologies is XML Schema, which provides a meta-level description of XML content. XML Schema can, in the simplest sense, be thought of as a means of dictating the format and content of XML documents. However, XML Schema's real power lies in the fact it can be used as a platform independent type description language, where XML documents are then used to transport data in accordance with those type descriptions.

XML technology is already well supported in terms of standard tools. In particular, the XML tools introduced here are widely available across platforms. While the specifics of using most XML tools may vary from platform-to-platform, the models are consistent which means that any experience with such tools is widely applicable.

The sum of these technologies means that XML is not only eminently expressive, but platform independent in the way it is written and processed. As we shall see, this is indeed a rich base on which to build interoperable systems. This is why Web services are based so heavily on XML.

Architect's Note

- XML is the single fundamental technology in Web services on which everything else is predicated. A good working knowledge of it will help you in the long run—where tools and toolkits fall short, you will be able to jump into the breach.
- Everything in the Web services architecture is governed by schemas. Every self-respecting architect and developer should understand XML Schemas, at least to the level presented here.
- XML Schemas are best used to describe type systems first and document layout second.
- When using XML within your own applications, make it a natural part of development to write schemas to accompany the documents. A good rule of thumb is: *A document is useless without its schema.*
- XML processing technologies are a commodity—don't reinvent the wheel unless you specifically cannot achieve your goals with off-the-shelf components.

CHAPTER 3

SOAP and WSDL

Web services are software components that expose their functionality to the network. To exploit that functionality, Web service consumers must be able to bind to a service and invoke its operations via its interface. To support this, we have two protocols that are the fundamental building blocks on which all else in the Web services arena is predicated: SOAP[1] and WSDL[2]. SOAP is the protocol via which Web services communicate, while WSDL is the technology that enables services to publish their interfaces to the network. In this chapter we cover both SOAP and WSDL in some depth and show how they can be used together with rudimentary tool support to form the basis of Web services-based applications.

The SOAP Model

Web services are an instance of the service-oriented architecture pattern that use SOAP as the (logical) transport mechanism for moving messages between services described by WSDL interfaces. This is a conceptually simple architecture, as shown in Figure 3-1, where SOAP messages are propagated via some underlying transport protocol between Web services.

1. In this chapter, unless otherwise explicitly stated, all references to SOAP and the SOAP Specification pertain to the SOAP 1.2 recommendation.
2. In this chapter, unless otherwise explicitly stated, all references to WSDL and the WSDL specification pertain to WSDL 1.1; see http://www.w3.org/TR/wsdl. The W3C's WSDL effort is less advanced than the latest SOAP work, though where possible we highlight new techniques from the WSDL 1.2 working drafts.

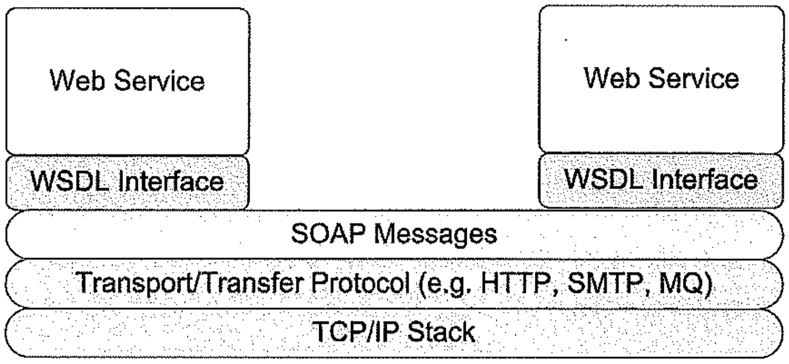


Figure 3-1 The logical Web services network.

A SOAP message is an XML document whose root element is called the *envelope*. Within the envelope, there are two child elements called the header and the body. Application payloads are carried in the body, while the information held in the header blocks usually contains data from the various Web services protocols that augment the basic SOAP infrastructure (and which is the primary subject of this book). The structure of a SOAP message is shown in Figure 3-2.

The SOAP message shown in Figure 3-2 provides the conceptual basis on which the whole SOAP model is based. Application payload travels in the body of the message and additional protocol messages travel in header blocks (which are optional, and may not be present if only application data is being transported). This permits a separation of concerns at the SOAP processing level between application-level messages and higher-level Web services protocols (e.g., transactions, security) whose payload travels in the SOAP header space.

The split between application and protocol data within SOAP messages allows the SOAP processing model to be a little more sophisticated than was suggested by the simple architecture shown in Figure 3-1. SOAP's distributed processing model outlines the fundamentals of the Web services architecture. It states (abstractly) how SOAP messages—including both the header and body elements—are processed as they are transmitted between Web services. In SOAP terms, we see that an application is comprised of *nodes* that exchange messages. The nodes are free to communicate in any manner they see fit, including any message-exchange pattern from one-way transmission through bilateral conversations. Furthermore, it is assumed in SOAP that messages may pass through any number of intermediate nodes between the sender and final recipient.

More interestingly however, the SOAP specification proposes a number of *roles* to describe the behavior of nodes under certain circumstances, which are shown in Figure 3-3. As a message progresses from node to node through a SOAP-based network, it encounters nodes that play the correct role for that message. Inside message elements, we may find role declarations that match these roles (or indeed other roles produced by third parties), and where we find a node and message part that match, the node executes its logic against the message. For example,

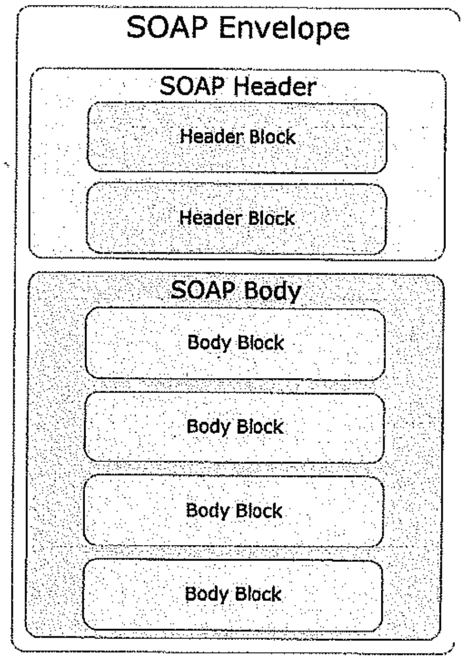


Figure 3-2 The structure of a SOAP message.

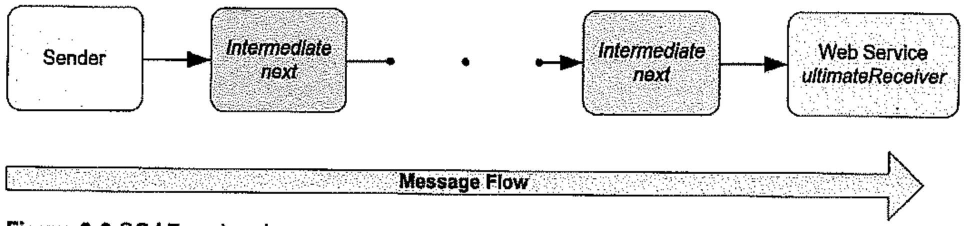


Figure 3-3 SOAP node roles.

where a node receives a message that specifies a role of `next` (and every node except the sender is always implicitly `next`), the node must perform the processing expected of that role or fault. In Figure 3-3, we see that the nodes labeled "intermediate" all play the role `next`. The Web service that finally consumes the message plays the role `ultimateReceiver`, and so each processes only the parts of the SOAP message which are (either implicitly or explicitly) marked as being for that role.

The processing model shown in Figure 3-3 is supported in software by SOAP servers. A SOAP server is a piece of middleware that mediates between SOAP traffic and application components, dealing with the message and processing model of the SOAP specification on a Web service's behalf. Therefore, to build Web services, it is important to understand how a SOAP server implements the SOAP model.

While it is impossible to cover every SOAP server platform here, we will examine the architecture of a generalized SOAP server (whose characteristics are actually derived from popular implementations such as Apache Axis and Microsoft ASP.Net) so that we have a mental model onto which we can hang various aspects of SOAP processing. An idealized view of a SOAP server is presented in Figure 3-4. This shows a generic SOAP server architecture. Inbound messages arrive via the physical network and are translated from the network protocol into the textual SOAP message. This SOAP message passes up the SOAP request stack where information stored in SOAP headers (typically context information for other Web services protocols like security, transactions and so forth) are processed by handlers that have been registered with the Web service. Such handlers are considered to be intermediate nodes in SOAP terms.

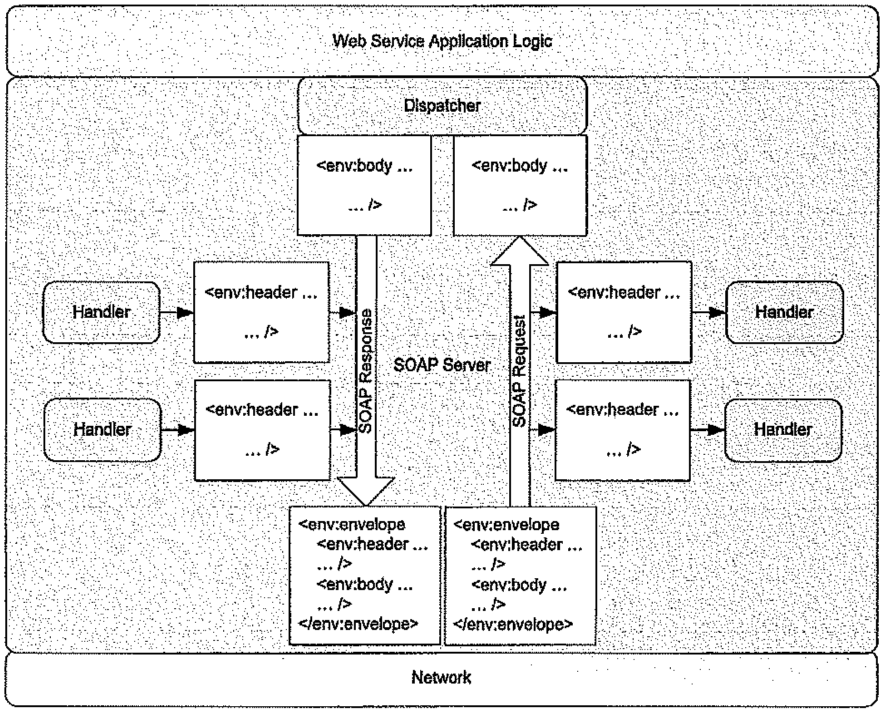


Figure 3-4 The architecture of a generalized SOAP server.

The handlers that operate on the headers are not generally part of the SOAP server by default, but are usually third-party components registered with the server to augment its capabilities. This means that SOAP servers are themselves extensible and can be upgraded to include additional protocol support over their lifetime as Web services' needs evolve.

At some later point, provided the header processing has not caused the service invocation to fail, the application payload of the message (carried in the SOAP body) reaches a dispatch mechanism where it causes some computation to occur within the back-end service implementation. The application logic then performs some computation before returning data to the dispatcher, which then propagates a SOAP message back down the SOAP response stack. Like the request stack, the response stack may have handlers registered with it which operate on the outgoing message, inserting headers into messages as they flow outward to be consumed by other Web services. Again, these handlers are considered to be nodes in SOAP terms.

Eventually, the outgoing message reaches the network level where it is marshaled into the appropriate network protocol and duly passes on to other SOAP nodes on the network, to be consumed by other SOAP nodes.

SOAP

Having understood the SOAP model and seen how this model is supported by SOAP servers, we can now begin to discuss the details of SOAP itself. SOAP is the oldest, most mature, and the single most important protocol in the Web services world. The SOAP specification defines this protocol as "[an] XML-based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses."[3]

In its earlier incarnations, the acronym SOAP used to stand for "Simple Object Access Protocol," though that meaning has ceased to exist in the SOAP 1.2 specification. This is undoubtedly a good thing since SOAP isn't especially simple, it's not exclusively designed for object access and it is more a packaging mechanism than a protocol per se.

In the following sections, we examine SOAP in some depth—from its basic use pattern and XML document structure, encoding schemes, RPC convention, binding SOAP messages, transport protocols, to using it as the basis for Web services communication.

3. http://www.w3.org/TR/SOAP/

SOAP Messages

We have already seen the overall structure of a SOAP message, as defined by the SOAP `Envelope`, in Figure 3-4. All SOAP messages, no matter how lengthy or complex, ultimately conform to this structure. The only caveat is there must be at least one body block within the SOAP body element in a message and there does not necessarily have to be a SOAP header or any SOAP header blocks. There is no upper limit on the number of header or body blocks, however. A sample SOAP message is presented here in Figure 3-5:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope" >
 <env:Header>
    <tx:transaction-id
        xmlns:tx="http://transaction.example.org"
      env:encodingStyle="http://transaction.example.org/enc"
        env:role=
"http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver"
        env:mustUnderstand="true">
           decd7461-4ef2138d-7b52e370-fed8a006-ca7ea17
    </tx:transaction-id>
 </env:Header>
 <env:Body xmlns:bank="http://bank.example.org">
    <bank:credit-account env:encodingStyle=
                   "http://www.w3.org/2002/06/soap-encoding">
        <bank:account>12345678</bank:account>
        <bank:sort>10-11-12</bank:sort>
        <bank:amount currency="usd">123.45</bank:amount>
    </bank:credit-account>
    <bank:debit-account>
        <bank:account>87654321</bank:account>
        <bank:sort>12-11-10</bank:sort>
        <bank:amount currency="usd">123.45</bank:amount>
    </bank:debit-account>
 </env:Body>
</env:Envelope>
```

Figure 3-5 A simple SOAP message.

The structure of all SOAP messages (including that shown in Figure 3-5) maps directly onto the abstract model shown in Figure 3-2. Figure 3-5 contains a typical SOAP message with a single header block (which presumably has something to do with managing transactional integrity), and a body containing two elements (which presumably instructs the recipient of the message to perform an operation on two bank accounts). Both the `Header` and `Body` elements are contained within the outer `Envelope` element, which acts solely as a container.

SOAP Envelope

The SOAP Envelope is the container structure for the SOAP message and is associated with the namespace http://www.w3.org/2002/06/soap-envelope. An example is shown in Figure 3-6 where the namespace is associated with the prefix env:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope" >
  <!-- Optional header blocks -->
  <env:Header>
    ...
  </env:Header>
  <!-- Single mandatory body element -->
  <env:Body xmlns:bank="http://bank.example.org">
    ...
  </env:Body>
</env:Envelope>
```

Figure 3-6 The SOAP envelope element.

The Envelope contains up to two child elements, the Header and the Body (where the Body element is mandatory). Aside from acting as a parent to the Header and the Body elements, the Envelope may also hold namespace declarations that are used within the message.

SOAP Header

The Header element provides a mechanism for extending the content of a SOAP message with out-of-band information designed to assist (in some arbitrary and extensible way) the passage of the application content in the Body section content through a Web services-based application.

The SOAP header space is where much of the value in Web services resides, since it is here that aspects like security, transactions, routing, and so on are expressed. Every Web services standard has staked its claim on some part of the SOAP header territory, but in a mutually compatible way. The fact that SOAP headers are extensible enough to support such diverse standards is a major win, since it supports flexible protocol composition tailored to suit specific application domains.

A SOAP header has the local name `Header` associated with the `http://www.w3.org/2002/06/soap-envelope` namespace. It may also contain any number of namespace qualified attributes and any number of child elements, known as *header blocks*. In the absence of any such header blocks, the `Header` element itself may be omitted from the `Envelope`. A sample header block is shown in Figure 3-7.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope" >
  <!-- Optional header blocks -->
  <env:Header>
    <tx:transaction-id
        xmlns:tx="http://transaction.example.org"
     env:encodingStyle="http://transaction.example.org/enc"
        env:role=
"http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver"
        env:mustUnderstand="true">
          decd7461-4ef2138d-7b52e370-fed8a006-ca7ea17
    </tx:transaction-id>
  </env:Header>

  <!-- Single mandatory body element -->
  <env:Body xmlns:bank="http://bank.example.org">
    ...
  </env:Body>
</env:Envelope>
```

Figure 3-7 A SOAP `header` element.

If present, each header block must be namespace qualified (according to the rules set out in the SOAP schema), may specify how it has been encoded (i.e., which schema constrains it) through the `encodingStyle` attribute, may specify its consumer through the `role` attribute, and may demand that it is understood by SOAP infrastructure that encounters its message through the `mustUnderstand` attribute. The SOAP specification stipulates that it is illegal for the `role` and `mustUnderstand` attributes to appear anywhere other than in header block declarations.

The sender of a SOAP message should not place them anywhere else, and a receiver of such a malformed message must ignore these attributes if they are out of place. These attributes are of fundamental importance to SOAP processing (and thus Web services) and warrant further discussion. The vehicle for this discussion is the example SOAP message shown in Figure 3-5 where we see a header block called `transaction-id` that provides the necessary out-of-band information for the application payload to be processed within a transaction (using a hypothetical transaction processing protocol).

The `role` Attribute

The `role` attribute controls the targeting of header blocks to particular SOAP nodes (where a SOAP node is an entity that is SOAP-aware). The `role` attribute contains a URI that identifies the `role` being played by the intended recipient of its header block. The SOAP node receiving the message containing the header block must check through the headers to see if any of the declared roles are applicable. If there are any matches, the header blocks must be processed or appropriate faults generated.

Although any URI is valid as a role for a SOAP node to assume, the SOAP specification provides three common roles that fit into the canonical SOAP processing model as part of the standard:

- `http://www.w3.org/2002/06/soap-envelope/role/none`: No SOAP processor should attempt to process this header block, although other header blocks may reference it and its contents, allowing data to be shared between header blocks (and thus save bandwidth in transmission).
- `http://www.w3.org/2002/06/soap-envelope/role/next`: Every node must be willing to assume this role since it dictates that header block content is meant for the next SOAP node in the message chain. If a node knows in advance that a subsequent node does not need a header block marked with the "next" role, then it is at liberty to remove that block from the header.
- `http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver`: The ultimate receiver is the final node in the message chain. Header blocks referencing this `role` attribute (or equivalently referencing no `role` attribute) should be delivered to this last node. It always implicitly plays the role of "next" given that the last node always comes after some other node—even in the simplest case where the last node comes immediately after the initiator.

Figure 3-8 highlights the role attribute from our example SOAP message in Figure 3-5:

```
<env:Header>
  <tx:transaction-id
      xmlns:tx="http://transaction.example.org"
    env:encodingStyle="http://transaction.example.org/enc"
      env:role=
"http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver"
      env:mustUnderstand="true">
         decd7461-4ef2138d-7b52e370-fed8a006-ca7ea17
  </tx:transaction-id>
</env:Header>
```

Figure 3-8 The `role` attribute.

The role attribute in Figure 3-8 has the value http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver, which means the contents of the header block are intended for the final SOAP processing node in this interaction (i.e., the recipient Web service). According to the SOAP processing model, this Web service must be capable of processing the application payload (in the SOAP body) in accordance with the transaction processing specification in the header block.

The mustUnderstand Attribute

If the mustUnderstand attribute is set to true, it implies that any SOAP infrastructure that receives the message containing that header block must be able to process it correctly or issue an appropriate fault message. Those header blocks that contain the mustUnderstand="true" attribute are known as *mandatory header blocks* since they must be processed by any nodes playing the matching roles. Header blocks missing their mustUnderstand attribute should still be examined by nodes that play the appropriate role. If a failure to act on a role occurs, it is not deemed to be critical and further processing may occur since by missing the mustUnderstand attribute they are not considered mandatory, as shown in Figure 3-9.

```
<env:Header>
  <tx:transaction-id
      xmlns:tx="http://transaction.example.org"
    env:encodingStyle="http://transaction.example.org/enc"
      env:role="http://www.w3.org/2002/06/soap-envelope/role/
ultimateReceiver"
      env:mustUnderstand="true">
         decd7461-4ef2138d-7b52e370-fed8a006-ca7ea17
  </tx:transaction-id>
</env:Header>
```

Figure 3-9 The mustUnderstand attribute.

The SOAP specification states that SOAP senders *should not* generate, but SOAP receivers *must* accept the SOAP mustUnderstand attribute information item with a value of "false" or "0". That is, a SOAP message should contain the literal values "true" and "false" in mustUnderstand attributes, not the characters "1" and "0".

In our example shown in Figure 3-9, the mustUnderstand attribute is set to true because it is imperative that the processing node must perform the account debit-credit within a transaction. If it cannot support transactional processing, then we would prefer that it leaves the accounts well alone—particularly if it is our money being transferred.

The encodingStyle Attribute

The `encodingStyle` attribute is used to declare how the contents of a header block were created. Knowing this information allows a recipient of the header to decode the information it contains. SOAP allows many encoding schemes and provides one of its own as an optional part of the spec. However, we will not dwell on such matters since this attribute is used not only in header blocks but in the body as well, and is covered in much more depth later in this chapter.

SOAP Body

In contrast to the intricacies of the SOAP header space, the body section of a SOAP envelope is straightforward, being simply a container for XML application payload. In fact the SOAP specification states "[T]his specification mandates no particular structure or interpretation of these elements, and provides no standard means for specifying the processing to be done." In our example in Figure 3-5, the application content housed by the SOAP `Body` consists of two elements that are interpreted as commands to debit and credit a bank account, which collectively amount to a funds transfer. The only constraints the SOAP specification places on the SOAP body are that it is implicitly targeted at the `ultimateRecipient` of the application content and that the ultimate recipient must understand its contents.

SOAP Faults

By contrast to its standard role as the simple carrier of application payload, the SOAP Body also acts in a far more interesting way as the conduit for propagating exceptions between the parties in a Web services application. The SOAP `Fault` is a reserved element predefined by the SOAP specification whose purpose is to provide an extensible mechanism for transporting structured and unstructured information about problems that have arisen during the processing of SOAP messages or subsequent application execution. Since the fault mechanism is predefined by the SOAP specification, SOAP toolkits are able to use this mechanism as a standard mechanism for distributed exception handling.

The SOAP `Fault` element belongs to the same namespace as the SOAP `Envelope` and contains two mandatory child elements: `Code` and `Reason`, and three optional elements: `Node`, `Role`, and `Detail`. An example of a SOAP Fault is shown in Figure 3-10 below. The fault is generated in response to the message shown in Figure 3-5 where the message conveyed information on a bank account cash transfer. To understand precisely what has caused the fault, we must understand each of the elements of which it is composed.

The first child element of the `Fault` is the `Code` element, which contains two subelements: a mandatory element called `Value` and an optional element called `Subcode`. The `Value` element can contain any of a small number of fault codes as qualified names (some-

```
<?xml version="1.0" ?>
<env:Envelope
     xmlns:env="http://www.w3.org/2002/06/soap-envelope"
     xmlns:bank="http://bank.example.org">
    <env:Body>
        <env:Fault>
            <env:Code>
                <env:Value>env:Receiver</env:Value>
                <env:Subcode>
                    <env:Value>bank:bad-account</env:Value>
                </env:Subcode>
            </env:Code>
            <env:Reason lang="en-UK">
             The specified account does exist at this branch
            </env:Reason>
            <env:Detail>
                <err:myfaultdetails
                  xmlns:err= "http://bank.example.org/fault">
                    <err:invalid-account-sortcode>
                        <bank:sortcode>
                            10-11-12
                        </bank:sortcode>
                        <bank:account>
                            12345678
                        </bank:account>
                    </err:invalid-account-sortcode >
                </err:myfaultdetails>
            </env:Detail>
        </env:Fault>
    </env:Body>
</env:Envelope>
```

Figure 3-10 An example SOAP fault.

times abbreviated to QName) from the `http://www.w3.org/2002/06/soap-envelope` namespace, as per Figure 3-11, where each QName identifies a reason why the fault was generated.

In Figure 3-10 the contents of the `env:Value` element is `env:Receiver` (shown in Figure 3-12), which tells us that it was the SOAP node at the end of the message path (the receiver) that generated the fault and not an intermediate node dealing with the transaction header block.

As shown in Figure 3-13, the `Subcode` element contains a `Value` element that gives application-specific information on the fault through the qualified name `bank:bad-account`. This QName has significance only within the scope of the application that issued it, and as such the `Subcode` mechanism provides the means for propagating finely targeted application-level exception messages.

Fault Code	Description
`VersionMismatch`	Occurs when SOAP infrastructure has detected mutually incompatible implementations based on different versions of the SOAP specification.
`MustUnderstand`	Issued in the case where a SOAP node has received a header block has with its `mustUnderstand` attribute set to true, but does not have the capability to correctly process that header block – that is, it does not understand the protocol with which that header block is associated.
`DataEncodingUnknown`	Arises when the content of either a header or body block is encoded according to a schema that the SOAP node reporting the fault does not understand.
`Sender`	Occurs when the sender propagated a malformed message, including messages with insufficient data to enable the recipient to process it. It is an indication that the message is not to be resent without change.
`Receiver`	Generated when the recipient of the SOAP message could not process the message content because of some application failure. Assuming the failure is transient, resending the message later may successfully invoke processing.

Figure 3-11 SOAP fault codes.

```
<env:Fault>
  <env:Code>
    <env:Value>env:Receiver</env:Value>
```

Figure 3-12 Identifying the faulting SOAP node.

```
<env:Subcode>
  <env:Value>bank:bad-account</env:Value>
</env:Subcode>
```

Figure 3-13 Application-specific fault information.

Though it isn't used in this fault, the Subcode element also makes the SOAP fault mechanism extensible. Like the Code element, the Subcode element also contains a mandatory Value child element and an optional Subcode element, which may contain further nested Subcode elements. The Value element of any Subcode contains a qualified name that consists of a prefix and a local name that references a particular QName within the application-level XML message set.

The Reason element associated with a Code is used to provide a human readable explanation of the fault, which in Figure 3-10 tells us that "The specified account does not exist at this branch". SOAP toolkits often use the contents of the Reason element when throwing exceptions or logging failures to make debugging easier. However, the Reason element is strictly meant for human consumption and it is considered bad practice to use its content for further processing.

The optional Node element provides information on which node in the SOAP message's path caused the fault. The content of the Node element is simply the URI of the node where the problem arose. In Figure 3-10 we do not have a Node element because it is the ultimate recipient of the message that caused the fault, and clearly the sender of the message already knows the URI of the recipient. However if, for example, an intermediate node dealing with the transactional aspects of the transfer failed, then we would expect that the Node element would be used to inform us of the intermediary's failure (and as we shall see, we would not expect a Detail element).

The Node element is complemented by the also optional Role element that provides information pertaining to what the failing node was doing at the point at which it failed. The Role element carries a URI that identifies the operation (usually some Web services standard) and that the party resolving the fault can use to determine what part of the application went wrong. Thus, the combination of Node and Role provides valuable feedback on exactly what went wrong and where.

The SOAP Detail element, as recapped in Figure 3-14, provides in-depth feedback on the fault if that fault was caused as a by-product of processing the SOAP Body.

```
<env:Detail>
  <err:myfaultdetails
    xmlns:err= "http://bank.example.org/fault">
    <err:invalid-account-sortcode>
      <bank:sortcode>
        10-11-12
      </bank:sortcode>
      <bank:account>
        12345678
      </bank:account>
    </err:invalid-account-sortcode >
  </err:myfaultdetails>
</env:Detail>
```

Figure 3-14 Fault detail in an application-specific form.

The presence of the `Detail` element provides information on faults arising from the application payload (i.e., the `Body` element had been at least partially processed by the ultimate recipient), whereas its absence indicates that the fault arose because of out-of-band information carried in header blocks. Thus we would expect that if a `Detail` block is present, as it is in Figure 3-10 and Figure 3-11, the `Node` and `Role` elements will be absent and vice versa.

The contents of the `Detail` element are known as *detail entries* and are application-specific and consist of any number of child elements. In Fault detail in an application-specific form. we see the `invalid-account-sortcode` element which describes the fault is some application specific fashion.

SOAP Encoding

The `encodingStyle` attribute appears in both header blocks and the body element of a SOAP message. As its name suggests, the attribute conveys information about how the contents of a particular element are encoded. At first this might seem a little odd since the SOAP message is expressed in XML. However, the SOAP specification is distinctly hands-off in specifying how header and body elements (aside from the SOAP `Fault` element) are composed, and defines only the overall structure of the message. Furthermore, XML is expressive and does not constrain the form of document a great deal and, therefore, we could imagine a number of different and mutually uninteroperable ways of encoding the same data, for example:

```
<account>
    <balance>
        123.45
    </balance>
</account>
```

and `<account balance="123.45"/>`
might both be informally interpreted in the same way by a human reader but would not be considered equivalent by XML processing software. Ironically, this is one of the downfalls of XML—it is so expressive that, given the chance, we would all express ourselves in completely different ways. To solve this problem, the `encodingStyle` attribute allows the form of the content to be constrained according to some schema shared between the sender and recipient.

One potential drawback is that senders and receivers of messages may not share schemas—indeed the senders and receivers may be applications that do not deal with XML at all—and thus the best intentions of a SOAP-based architecture may be laid to waste. To avoid such problems, the SOAP specification has its own schema and rules for converting application-level data into a form suitable for embedding into SOAP messages. This is known as SOAP Encoding, and is associated with the namespace `env:encodingStyle="http://www.w3.org/2002/06/soap-encoding"`.

The rules for encoding application data as SOAP messages are captured in the SOAP specification as the SOAP Data Model. This is a straightforward and concise part of the specification that describes how to reduce data structures to a directed, labeled graph. While it is outside of the scope of this book to detail the SOAP Data Model, the general technique is shown in Figure 3-15. This SOAP encoding example, highlights the fact that there are two aspects to the encoding. The first of these is to transform a data structure from an application into a form suitable for expressing in XML via the rules specified in the SOAP Data Model. The other aspect is to ensure that all the data in the subsequent XML document is properly constrained by the SOAP schema. It is worth noting that SOAP provides low entry point through SOAP encoding since a SOAP toolkit will support the serialization and deserialization of arbitrary graphs of objects via this model, with minimal effort required of the developer. In fact, coupled with the fact that SOAP has a packaging mechanism for managing message content, and a means (though SOAP encoding) of easily creating message content we are close to having an XML-based Remote Procedure Call mechanism.

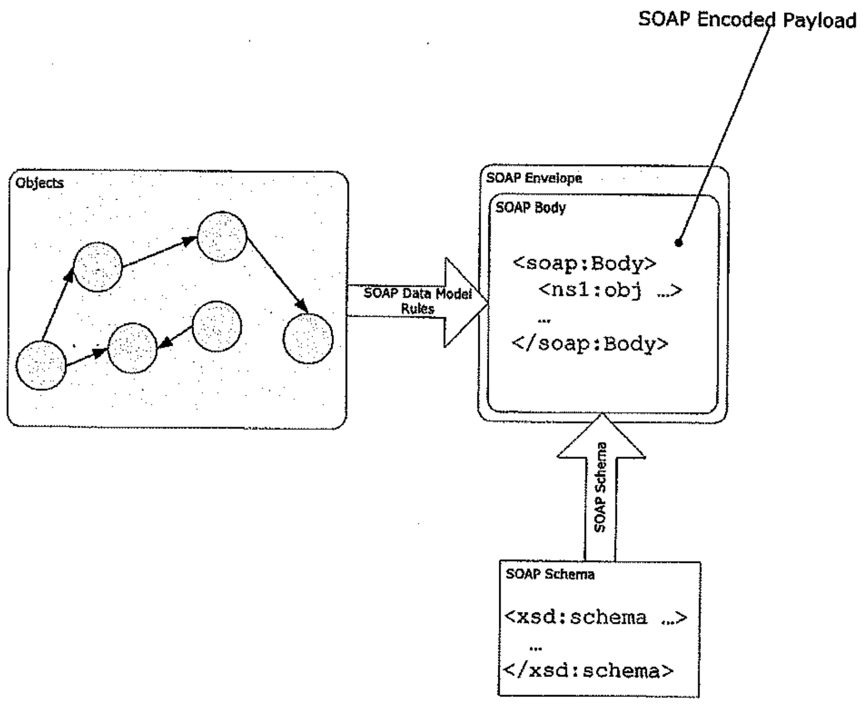


Figure 3-15 SOAP encoding application-level objects.

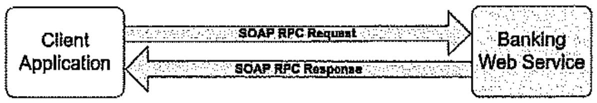
SOAP RPC

As it happens, the SOAP specification is useful straight "out of the box." The fact that it provides both a message format and marshalling naturally supports RPC, and indeed millions of developers worldwide will by now have seen how easy it is to run SOAP RPC-based Web services on a myriad of platforms. It's probably not the case that SOAP RPC will be the dominant paradigm for SOAP in the long term, but it is easy to achieve results with SOAP RPC quickly because all the major toolkits support it and RPC is a pattern many developers are familiar with.

Note that although SOAP RPC has enjoyed some prominence in older Web services toolkits, there is a majority consensus of opinion in the Web services community that more coarse-grained, document-oriented interactions should be the norm when using SOAP.

SOAP RPC provides toolkits with a convention for packaging SOAP-encoded messages so they can be easily mapped onto procedure calls in programming languages. To illustrate, let's return to our banking scenario and see how SOAP RPC might be used to expose account management facilities to users. Bear in mind throughout this simple example that it is an utterly insecure instance whose purpose is to demonstrate SOAP RPC only.

Figure 3-16 shows a simple interaction between a Web service that offers the facility to open bank accounts and a client that consumes this functionality on behalf of a user. The Web service supports an operation called `openAccount(...)` which it exposes through a SOAP server and advertises as being accessible via SOAP RPC (SOAP does not itself provide a means of describing interfaces, but as we shall see later in the chapter, WSDL does). The client inter-



```
// Client implementation
Bank b = new Bank();
String account =
b.openAccout(title,
                    surname,
                    firstname,
                    postcode,
                    telephone);
```

```
// Service implementation
class Bank
{
    public String openAccout(title,
                    surname,
                    firstname,
                    postcode,
                    telephone)
    {
        // Some account processing
        return accoutNumber;
    }
}
```

Figure 3-16 Interacting with a banking service via SOAP RPC.

acts with this service through a stub or proxy class called `Bank` which is toolkit-generated (though masochists are free to generate their own stubs) and deals with the marshalling and unmarshalling of local variables into SOAP RPC messages.

In this simple use case, the SOAP on the wire between the client and Web service is similarly straightforward. Figure 3-17 shows the SOAP RPC request sent from the client to the Web service:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope">
    <env:Body>
        <bank:openAccount env:encodingStyle=
         "http://www.w3.org/2002/06/soap-encoding"
         xmlns:bank="http://bank.example.org/account"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <bank:title xsi:type="xs:string">
                Mr
            </bank:title>
            <bank:surname xsi:type="xs:string">
                Bond
            </bank:surname>
            <bank:firstname xsi:type="xs:string">
                James
            </bank:firstname>
            <bank:postcode xsi:type="xs:string">
                S1 3AZ
            </bank:postcode>
            <bank:telephone xsi:type="xs:string">
                09876 123456
            </bank:telephone>
        </bank:openAccount>
    </env:Body>
</env:Envelope>
```

Figure 3-17 A SOAP RPC request.

There is nothing particularly surprising about the RPC request presented in Figure 3-17. As per the RPC specification, the content is held entirely within the SOAP body (SOAP RPC does not preclude the use of header blocks, but they are unnecessary for this example), and the name of the element (`openAccount`) matches the name of the method to be called on the Web service. The contents of the `bank:openAccount` correspond to the parameters of the `openAccount` method shown in Figure 3-16, with the addition of the `xsi:type` attribute to help recipients of the message to convert the contents of each parameter element to the correct kind of variable in specific programming languages. The response to the original request follows a slightly more intricate set of rules and conventions as shown in Figure 3-18.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
     xmlns:env="http://www.w3.org/2002/06/soap-envelope">
    <env:Body>
        <bank:openAccountResponse env:encodingStyle=
         "http://www.w3.org/2002/06/soap-encoding" xmlns:rpc=
         "http://www.w3.org/2002/06/soap-rpc" xmlns:bank=
         "http://bank.example.org/account" xmlns:xs=
         "http://www.w3.org/2001/XMLSchema" xmlns:xsi=
          "http://www.w3.org/2001/XMLSchema-instance">
            <rpc:result>bank:accountNo</rpc:result>
            <bank:accountNo xsi:type="xsd:int">
                10000014
            </bank:accountNo>
        </bank:openAccountResponse>
    </env:Body>
</env:Envelope>
```

Figure 3-18 A SOAP RPC response.

The SOAP RPC response is slightly more complex and interesting than the request, and there are two noteworthy aspects of the SOAP RPC response. The first is that by convention the name of the response element is the same as the request element with *Response* appended (and toolkits make use of this convention so it's practically standard now).

The second interesting aspect is that the response is capable of matching the procedure call semantics of many languages since it supports in, out, and in/out parameters as well as return values where an "in" parameter sources a variable to the procedure call; an "out" parameter sources nothing to the procedure but is populated with data at the end of the procedure call. An "in/out" parameter does both, while a return value is similar to an out parameter with the exception that its data may be ignored by the caller.

In this example, we have five "in" parameters (title, surname, first name, post code, and telephone number) which we saw in the SOAP request and expect a single return value (account number) which we see in the SOAP response. The return value is also interesting because, due to its importance in most programming languages, it is separated from out and in/out parameters by the addition of the `rpc:result` element that contains a `QName` that references the element which holds the return value. Other elements which are not referenced are simply treated as out or in/out parameters. This behavior is different from previous versions of SOAP where the return value was distinguished by being first among the child elements of the response. This was rectified for SOAP 1.2 because of the inevitable ambiguity that such a contrivance incurs—what happens when there is no return value?

Of course in a textbook example like this, everything has worked correctly and no problems were encountered. Indeed, you would be hard pressed to find a reader who would enjoy a book where the examples were a set of abject failures. However, like paying taxes and dying, computing systems failures seem inevitable. To cover those cases where things go wrong, SOAP

RPC takes advantage of the SOAP fault mechanism with a set of additional fault codes (whose namespace is `http://www.w3.org/2002/06/soap-rpc`), which are used in preference to the standard SOAP fault codes in RPC-based messages shown in Figure 3-19, in decreasing order of precedence.

Fault	SOAP Encoding for Fault
Transient fault at receiver (e.g. out of memory error).	Fault with value of `env:Receiver` should be generated.
Receiver does not understand data encoding (e.g. encoding mechanism substantially different at sender and receiver)	A fault with a `Value` of `env:DataEncodingUnknown` for `Code` should be generated.
The service being invoked does not expose a method matching the name of the RPC element.	A fault with a `Value` of `env:Sender` for `Code` and a `Value` of `rpc:ProcedureNotPresent` for `Subcode` may be generated.
The receiver cannot parse the arguments sent. There may be too many or too few arguments, or there may be type mismatches.	A fault with a `Value` of `env:Sender` for `Code` and a `Value` of `rpc:BadArguments` for `Subcode` must be generated.

Figure 3-19 SOAP RPC faults.

Finally, in Figure 3-20 we see a SOAP RPC fault in action where a poorly constructed client application has tried to invoke an operation on the bank Web service, but has populated its request message with nonsense. In this figure, the bank Web service responds with a SOAP RPC fault that identifies the faulting actor (the sender) as part of the `Code` element. It also describes what the faulting actor did wrong (sent bad arguments) by specified the `QName rpc:BadArguments` as part of the `subcode` element. It also contains some human-readable information to aid debugging (missing surname parameter), in the `Reason` element.

```
<?xml version="1.0"?>
<env:Envelope
     xmlns:env="http://www.w3.org/2002/06/soap-envelope"
     xmlns:rpc="http://www.w3.org/2002/06/soap-rpc">
    <env:Body>
        <env:Fault>
            <env:Code>
                <env:Value>env:Sender</env:Value>
                <env:Subcode>
                    <env:Value>rpc:BadArguments</env:Value>
                </env:Subcode>
            </env:Code>
            <env:Reason>
                Missing surname parameter
            </env:Reason>
        </env:Fault>
    </env:Body>
</env:Envelope>
```

Figure 3-20 A SOAP RPC fault.

Using Alternative SOAP Encodings

Of course some applications already deal with XML natively, and there are currently XML-based vocabularies in use today supporting a plethora of B2B applications. SOAP-based messaging can take advantage of the pre-existence of schemas to craft message exchanges that compliment existing systems using so-called document-style SOAP.

The way in which alternative SOAP encodings are handled is straightforward. Instead of encoding header or body content according to the SOAP Data Model, we simply encode according to the rules and constraints of our data model and schema. In essence, we can just slide our own XML documents into a SOAP message, providing we remember to specify the `encodingStyle` attribute (and of course ensuring that the intended recipients of the message can understand it). This style of SOAP encoding is known as literal style and naturally suits the interchange of business-level documents based on their existing schemas.

This is a definite boon to SOAP use and by our estimation, the future dominant paradigm for SOAP use. Its plus points include not only the ability to re-use existing schemas, but by dint of the fact that we are now dealing with message exchanges and not remote procedure calls, we are encouraged to design Web service interactions with much coarser granularity. In essence, we are changing from a fine-grained model that RPC encourages (you send a little bit of data, get a little bit back and make further calls until your business is completed), to a much coarser-grained model where you send all the data necessary to get some business process done at the recipient end, and expect that the recipient may take some time before he gets back to you with a complete answer.

One particularly apt view of the fine- versus coarse-grained view of Web services interactions is that of a phone call versus a facsimile transmission.[4] Where we interact with a business over the phone there is a great deal of back-and-forth between ourselves and the agent of the business to whom we are talking. We both have to establish contexts and roles for each other, and then enter into a socially and linguistically complex conversation to get business transacted. Small units of data are exchanged like "color" and "amount" that are meaningless without the context, and if the call is lost we have to start over. This is fine-grained interaction.

While we would not seek to undermine the value of good old human-to-human communication, sometimes we just don't have time for this. It's even worse for our computing systems to have to communicate this way since they don't get any of the social pleasures of talking to each other. A better solution is often to obtain a catalog or brochure for the business that we want to trade with. When we have the catalog, we can spend time pouring over the contents to see what goods or services we require. Once we are certain of what we want, we can just fill in and fax the order form to the company and soon our products arrive via postal mail.

This system is eminently preferable for business processes based on Web services. For a start, complex and meaningful data was exchanged that does not rely on context. A catalog and an order form are descriptive enough to be universally understood and the frequency of data exchange was low, which presents less opportunity for things to go astray. This system is also loosely coupled when the systems are not directly communicating (which only happens twice: once for catalog delivery and once while the order is being faxed). They are not affected by one another and do not tie up one another's resources—quite the contrary to the telephone-based system.

Of course, we don't necessarily get something for nothing. The price that we must pay as developers is that we must write the code to deal with the encoding schemes we choose. In the SOAP RPC domain where the encoding is fixed and the serialization from application-level data structure to XML is governed by the SOAP Data Model, toolkits could take care of much of this work. Unfortunately, when we are working with our own schemas, we cannot expect SOAP toolkits to be able to second-guess its semantics and, thus, we have to develop our own handler code to deal with it, as shown in Figure 3-21.

4 See "The 7 Principles of Web Services Business Process Management" at http://www.iona.com/whitepapers/Principles-of-Web-Services-and-BPM.pdf

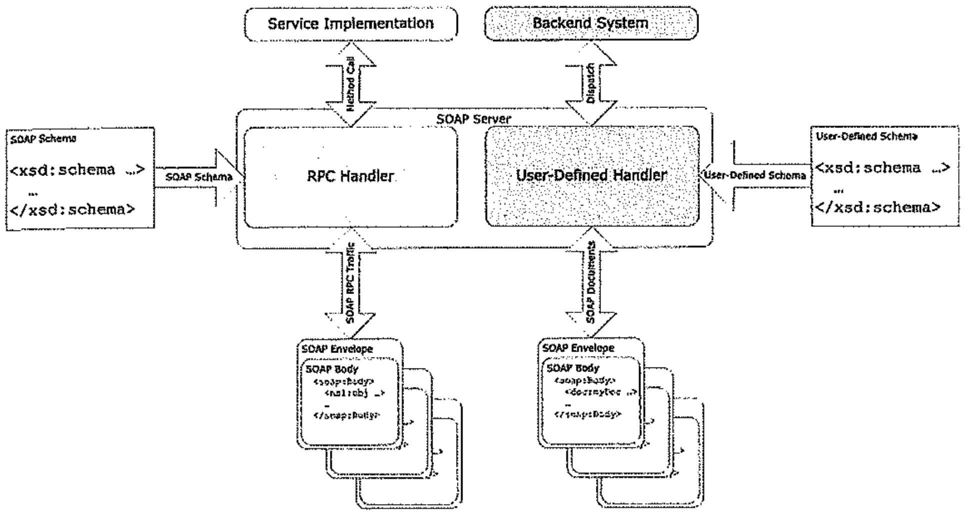


Figure 3-21 Document-oriented SOAP processing.

The user-defined handler in Figure 3-21 is one of potentially many handlers deployed onto the SOAP server to provide the functionality to deal with SOAP messages encoded with arbitrary schemas. Where the SOAP RPC handler simply dispatches the contents of the SOAP RPC messages it receives to appropriate method calls, there are no such constraints on a Web service which uses document-style SOAP. One valid method would be to simply pick out the important values from the incoming document and use them to call a method, just like the SOAP RPC handler. However, as more enterprises become focused on XML as a standard means for transporting data within the enterprise boundary, it is more likely that the contents of the SOAP body will flow directly onto the intranet. Once delivered to the intranet, the messages may be transformed into proprietary XML formats for inclusion with in-house applications, or may be used to trigger business processes without the need to perform the kind of marshalling/unmarshalling required for SOAP RPC.

Note that irrespective of whether the application payload in SOAP messages is SOAP-encoded, or encoded according to a third-party schema, the way that header blocks are used to convey out-of-band information to support advanced Web services protocols is unaffected. Headers are an orthogonal issue to the application-level content.

Document, RPC, Literal, Encoded

Much of the confusion in understanding SOAP comes from the fact that several of the key terms are overloaded. For example, both SOAP RPC (meaning the convention for performing remote procedure calls via SOAP) and RPC style are both valid pieces of SOAP terminology. In this section we clarify the meaning of each of these terms so they do not cause further confusion as we begin discussing WSDL.

Document

Document-style SOAP refers to the way in which the application payload is hosted within the SOAP `Body` element. When we use document style, it means the document is placed directly as a child of the `Body` element, as shown on the left in Figure 3-22, where the application content is a direct child of the `<soap:Body>` element.

```
          Document                          RPC
        ------------------------------------------------------------
        <soap:Body>                         <soap:Body>
Literal   <inv:invoice ...>                   <m:purchase>
            <inv:orderNo ...                    <inv:invoice ...>
          </inv:invoice>                          <inv:orderNo ...
        </soap:Body>                            </inv:invoice>
                                              </m:purchase>
                                            </soap:Body>

        <soap:Body>                         <soap:Body>
Encoded   <ns1:invoice>                       <m:purchase>
            <ns1:orderNo...                     <ns1:invoice>
          </ns1:invoice>                          <ns1:orderNo...
        </soap:Body>                            </ns1:invoice>
                                              </m:purchase>
                                            </soap:Body>
```

Figure 3-22 Document, RPC, Literal, and Encoded SOAP messages.

RPC

RPC-style SOAP wraps the application content inside an element whose name can be used to indicate the name of a method to dispatch the content to. This is shown on the right-hand side of Figure 3-22, where we see the application content wrapped `<m:purchase>` element.

Literal

Literal SOAP messages use arbitrary schemas to provide the meta-level description (and constraints) of the SOAP payload. Thus when using literal SOAP, we see that it is akin to taking an instance document of a particular schema and embedding it directly into the SOAP message, as shown at the top of Figure 3-22.

Encoded

SOAP-encoded messages are created by transforming application-level data structures via the SOAP Data Model into a XML format that conforms to the SOAP Schema. Thus, encoded messages tend to look machine-produced and may not generally resemble the same message expressed as a literal. Encoded messages are shown at the bottom of Figure 3-22.

SOAP RPC and SOAP Document-Literal

The SOAP specification provides four ways in which we could package SOAP messages, as shown in Figure 3-22. However, in Web services we tend to use only two of them: SOAP encoded-rpc (when combined with a request-response protocol becomes the SOAP RPC convention) and SOAP document-literal.

Document-literal is the preferred means of exchanging SOAP messages since it just packages application-level XML documents into the SOAP `Body` for transport without placing any semantics on the content.

As we have previously seen, with SOAP RPC the implied semantics are that the first child of the SOAP `Body` element names a method to which the content should be dispatched.

The remaining two options, document-encoded and rpc-literal, are seldom used since they mix styles to no great effect. Encoding documents is pointless if we already have schemas that describe them. Similarly, wrapping a document within a named element is futile unless we are going to use that convention as a remote procedure call mechanism. Since we already have SOAP RPC, this is simply a waste of effort.

SOAP, Web Services, and the REST Architecture

The World Wide Web (WWW) is unquestionably the largest and, by implication, the most scalable distributed system ever built. Though its original goal of simple content delivery was modest, the way that the Web has scaled is nothing short of miraculous.

Given the success of the Web, there is a body of opinion involved in designing the fundamental Web services architecture (that includes the SOAP specification) for which the means to achieving the same level of application scalability through Web services mirrors that of content scalability in the WWW.

The members of this group are proponents of the REST (*REpresentational State Transfer*) architecture, which it is claimed is "Web-Friendly." The REST architecture sees a distributed system as a collection of named resources (named with URIs) that support a small set of common verbs (GET, PUT, POST, DELETE) in common with the WWW.

The REST idea of defining global methods is similar to the UNIX concept of pipelining programs. UNIX programs all have three simple interfaces defined (STDIN, STDOUT, STDERR) for every program, which allows any two arbitrary programs to interact. The simplicity of REST as compared to custom network interfaces is analogous to the simplicity of UNIX pipelines vs. writing a custom application to achieve the same functionality. REST embraces simplicity and gains scalability as a result. The Web is a REST system, and the generic interfaces in question are completely described by the semantics of HTTP.[5]

What this means to the SOAP developer is that certain operations involving the retrieval of data without changing the state of a Web resource should be performed in a manner that is harmonious with the Web. For example, imagine that we want to retrieve the balance of our account from our bank Web service. Ordinarily we might have thought that something like that shown in Figure 3-23 would be ideal. If this message was sent as part of a HTTP POST, then it would be delivered to the SOAP server, which would then extract the parameters and deliver the results via the `getBalanceResponse` message.

```
<?xml version="1.0" ?>
<env:Envelope
     xmlns:env="http://www.w3.org/2002/06/soap-envelope" >
  <env:Body>
    <bank:getBalance
  env:encodingStyle="http://www.w3.org/2002/06/soap-encoding"
  xmlns:bank="http://bank.example.org/">
      <bank:accountNo>
           12345678
      </bank:accountNo>
    </bank:getBalance>
  </env:Body>
</env:Envelope>
```

Figure 3-23 A "Web-Unfriendly" message.

5. See RESTwiki, http://internet.conveyor.com/RESTwiki/moin.cgi/FrontPage

However, this is now discouraged by the SOAP specification and instead we are encouraged to use HTTP directly to retrieve the information, rather than "tunneling" SOAP through HTTP to get the information. A "Web-friendly" equivalent of Figure 3-23 is shown in Figure 3-24 where the HTTP request directly identifies the information to be retrieved and informs the Web service that it wants the returned information delivered in SOAP format.

```
GET /account?no=12345678  HTTP/1.1
Host: bank.example.org
Accept: application/soap+xml
```

Figure 3-24 A "Web-Friendly" message.

Figure 3-24 is certainly Web friendly since it uses the Web's application protocol (HTTP). However, there are a number of obstacles that have not yet been overcome at the time of writing that may prove detrimental to this approach:

1. A service may be exposed over other protocols than HTTP (e.g., SMTP that does not support the GET verb).
2. This scheme cannot be used if there are intermediate nodes that process SOAP header blocks.
3. There is no guidance yet provided by the SOAP specification authors on how to turn an RPC definition into its Web-friendly format.
4. Too much choice for little gain since we have to support the "Web-Unfriendly" approach anyway for those interactions that require the exchange of structured data.

While these techniques may yet come to fruition, it may be a long time before resolution is reached. When architecting applications today, the best compromise that we can offer is to be aware of those situations where you are engaged in pure information retrieval, and ensure that your architecture is extensible enough to change to a Web-friendly mechanism for those interactions tomorrow. Make sure the code that deals with Web services interactions is modular enough to be easily replaced by Web-friendly modules when the W3C architectural recommendations become more specific.

Looking Back to SOAP 1.1

While Web services will migrate toward SOAP 1.2 in the near future, the most prevalent Web services technology today is the now deprecated SOAP 1.1. Although there isn't a great deal that has changed between the two revisions, there are some caveats we must be aware of when dealing with SOAP 1.1-based systems. To ensure that the work we've invested in understanding

SOAP 1.2 isn't lost on SOAP 1.1 systems, we shall finish our coverage of SOAP with a set of notes that should make our SOAP 1.2 knowledge backwardly compatible with SOAP 1.1.[6]

Syntactic Differences between SOAP 1.2 and SOAP 1.1

- SOAP 1.2 does not permit any element after the body. The SOAP 1.1 schema definition allowed for such a possibility, but the textual description is silent about it. However, the Web Services Interoperability Organization (WS-I) has recently disallowed this practice in its basic profile and as such we should now consider that no elements are allowed after the SOAP body, since any other interpretation will hamper interoperability.
- SOAP 1.2 does not allow the `encodingStyle` attribute to appear on the SOAP Envelope, while SOAP 1.1 allows it to appear on any element.
- SOAP 1.2 defines the new `Misunderstood` header element for conveying information on a mandatory header block that could not be processed, as indicated by the presence of a `mustUnderstand` fault code. SOAP 1.1 provided the fault code, but no details on its use.
- In the SOAP 1.2 infoset-based description, the `mustUnderstand` attribute in header elements takes the (logical) value `true` or `false` while in SOAP 1.1 they are the literal value `1` or `0`, respectively.
- SOAP 1.2 provides a new fault code `DataEncodingUnknown`.
- The various namespaces defined by the two protocols are different.
- SOAP 1.2 replaces the attribute `actor` with `role` but with essentially the same semantics.
- SOAP 1.2 defines two new roles, `none` and `ultimateReceiver`, together with a more detailed processing model on how these behave.
- SOAP 1.2 has removed the dot notation for fault codes, which are now simply of the form `env:name`, where `env` is the SOAP envelope namespace.
- SOAP 1.2 replaces `client` and `server` fault codes with `Sender` and `Receiver`.
- SOAP 1.2 uses the element names `Code` and `Reason`, respectively, for what is called `faultcode` and `faultstring` in SOAP 1.1.
- SOAP 1.2 provides a hierarchical structure for the mandatory SOAP `Code` element, and introduces two new optional subelements, `Node` and `Role`.

6. These notes are abridged from the SOAP 1.2. Primer document which can be found at: http://www.w3.org/TR/2002/WD-soap12-part0-20020626/

Changes to SOAP-RPC

Though there was some feeling in the SOAP community that SOAP RPC has had its day and should be dropped in favor of a purely document-oriented protocol, the widespread acceptance of SOAP RPC has meant that it persists in SOAP 1.2, but with a few notable differences:

- SOAP 1.2 provides a `rpc:result` element accessor for RPCs.
- SOAP 1.2 provides several additional fault codes in the RPC namespace.
- SOAP 1.2 allows RPC requests and responses to be modeled as both structs as well as arrays. SOAP 1.1 allowed only the former construct.
- SOAP 1.2 offers guidance on a Web-friendly approach to defining RPCs where the method's purpose is purely a "safe" informational retrieval.

SOAP Encoding

Given the fact that SOAP RPC is still supported in SOAP 1.2 and that there have been some changes to the RPC mechanism, some portions of the SOAP encoding part of the specification have been updated to either better reflect the changes made to SOAP RPC in SOAP 1.2, or to provide performance enhancements compared to their SOAP 1.1 equivalents.

- An abstract data model based on a directed edge-labeled graph has been formulated for SOAP 1.2. The SOAP 1.2 encodings are dependent on this data model. The SOAP RPC conventions are dependent on this data model, but have no dependencies on the SOAP encoding. Support of the SOAP 1.2 encodings and SOAP 1.2 RPC conventions are optional.
- The syntax for the serialization of an array has been changed in SOAP 1.2 from that in SOAP 1.1.
- The support provided in SOAP 1.1 for partially transmitted and sparse arrays is not available in SOAP 1.2.
- SOAP 1.2 allows the inline serialization of multi-ref values.
- The `href` attribute in SOAP 1.1 of type `anyURI`, is called `ref` in SOAP 1.2 and is of type `IDREF`.
- In SOAP 1.2, omitted accessors of compound types are made equal to NILs.
- SOAP 1.2 provides several fault subcodes for indicating encoding errors.
- Types on nodes are made optional in SOAP 1.2.

While most of these issues are aimed at the developers of SOAP infrastucture, it is often useful to bear these features in mind for debugging purposes, especially while we are in the changeover period before SOAP 1.2 becomes the dominant SOAP version.

WSDL

Having a means of transporting data between Web services is only half the story. Without interface descriptions for our Web services, they are about as useful as any other undocumented API—very little! While in theory we could simply examine the message schemas for a Web service and figure out for ourselves how to interoperate with it, this is a difficult and error-prone process and one which could be safely automated if Web services had recognizable interfaces. Fortunately, WSDL provides this capability and more for Web services.

The Web Service Description Language or WSDL—pronounced "Whiz Dull"—is the equivalent of an XML-based IDL from CORBA or COM, and is used to describe a Web service's endpoints to other software agents with which it will interact. WSDL can be used to specify the interfaces of Web services bound to a number of protocols including HTTP GET and POST, but we are only interested in WSDL's SOAP support here, since it is SOAP which we consider to support the (logical) Web services network. In the remainder of this chapter we explore WSDL and show how we can build rich interfaces for Web services that enable truly dynamic discovery and binding, and show how WSDL can be used as the basis of other protocols and extended to other domains outside of interface description.

WSDL Structure

A WSDL interface logically consists of two parts: the abstract parts that describe the operations the Web service supports and the types of messages that parameterize those operations; and the concrete parts that describe how those operations are tied to a physical network endpoint and how messages are mapped onto specific carrier protocols which that network endpoint supports. The general structure of a WSDL document is shown in Figure 3-25.

The foundation of any WSDL interface is the set of messages that the service behind the interface expects to send and receive. A `message` is normally defined using XML Schema types (though WSDL allows other schema languages to be used) and is partitioned into a number of logical parts to ease access to its contents.

Messages themselves are grouped into WSDL `operation` elements that have similar semantics to function signatures in an imperative programming language. Like a function signature, an operation has input, output, and fault messages where WSDL supports at most a single input and output message, but permits the declaration of an arbitrary number of faults.

The `portType` is where what we think of as a Web service begins to take shape. A `portType` is a collection of operations that we consider to be a Web service. However, at this point the operations are still defined in abstract terms, simply grouping sets of message exchanges into operations.

The `binding` section of a WSDL interface describes how to map the abstractly defined messages and operations onto a physical carrier protocol. Each `operation` from each `portType` that is to be bound to a specific protocol (and thus ultimately be made available to the net-

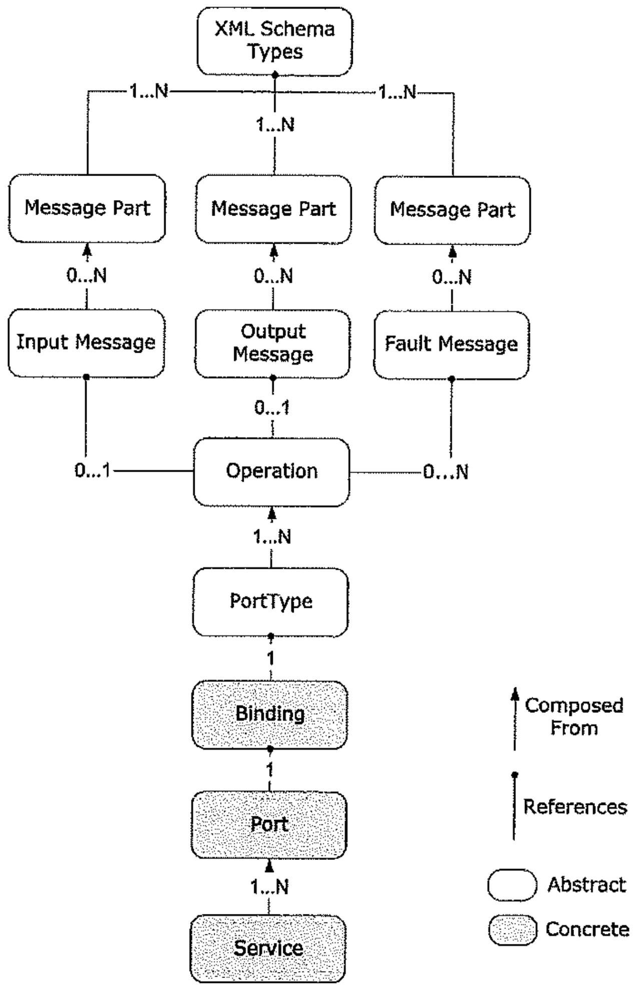


Figure 3-25 WSDL structure.

work) is augmented with binding information from the binding part of the WSDL specification—WSDL supports SOAP, HTTP GET and POST, and MIME—to provide a protocol-specific version of the original `portType` declaration.

Finally, a `port` is declared that references a particular `binding`, and along with addressing information is wrapped together into a `service` element to form the final physical, network addressable Web service.

As we saw in Figure 3-25, the abstract components of a WSDL description are the `types`, `message`, and `portType` elements, while the concrete elements are `binding` and `service`.

The split between abstract and concrete is useful, because it allows us to design interfaces in isolation from eventual deployment environments, using only the abstract definitions in WSDL. Once we are happy with the abstract aspects of the Web service interface, we can then write the concrete parts to tie the service down to a specific location, accessible over a specific protocol.

The Stock Quote WSDL Interface

Having seen WSDL from a theoretical perspective, we can concretize that theory by considering a specific example. The classic Web services application is the stock ticker example where a Web service provides stock quotes on request. Throughout the remainder of this discussion, we shall use a simple Web service which supports a single operation that has an equivalent signature to the following C# code:

```
double GetStockQuote(string symbol);
```

We examine WSDL stage by stage and show how we can turn this simple method signature into a true Web service interface.

Definitions

The opening element of any WSDL document is `definitions`, which is the parent for all other elements in the WSDL document. As well as acting as a container, the `definitions` element is also the place where global namespace declarations are placed.

```
<wsdl:definitions
  targetNamespace="http://stock.example.org/wsdl"
  xmlns:tns="http://stock.example.org/wsdl"
  xmlns:stockQ="http://stock.example.org/schema"
  xmlns:wsdl="http://www.w3.org/2003/02/wsdl">
  <!-- Remainder of WSDL description omitted -->
</wsdl:definitions>
```

Figure 3-26 The WSDL `definitions` element.

A typical WSDL `definitions` element takes the form shown in Figure 3-26, where the element declares the target namespace of the document, a corresponding prefix for that namespace, and a namespace prefix for the WSDL namespace (or alternatively it is also common to use the WSDL namespace as the default namespace for the whole document). Other namespaces may also be declared at this scope, or may be declared locally to their use within the rest of the document. Good practice for declaring namespaces to WSDL documents is to ensure the namespaces that are required for the abstract parts of the document are declared at this level, while namespaces required for the concrete parts of a WSDL document (like the bindings section) are declared locally to make factoring and management of WSDL documents easier.

The Types Element

The `types` element is where types used in the interface description are defined, usually in XML Schema types, since XML Schema is the recommended schema language for WSDL. For instance in our simple stock quote Web service, we define types that represent traded stocks and advertise those types as part of its WSDL interface as illustrated in Figure 3-27.

```
<wsdl:definitions … >
  <wsdl:import namespace="http://stock.example.org/schema"
    location="http://stock.example.org/schema"/>
  <wsdl:types xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="stock-quote">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="symbol" ref="stockQ:symbol"/>
          <xs:element name="lastPrice" ref="stockQ:price"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <!-- Other schema type definitions -->
  <wsdl:types>
</wsdl:definitions>
```

Figure 3-27 Defining `types` in a WSDL interface.

Before writing the `types` section, we first `import` some types declared by an external schema that make the types within that schema available to this WSDL document to build on. Those schema types (symbol and price) are used to create a new complex type (`stock-price`) which the WSDL interface will use to advertise its associated Web service.

The orthodox view is to use XML Schema to provide constraints and type information in Web services-based applications. However it is not necessarily the case that XML Schema is the right choice for every application domain, particularly those domains that have already chosen a different schema language on which to base their interoperability infrastructure. Recognizing this requirement, WSDL 1.2 supports the notion of other schema languages being used in place of the recommended XML Schema. Although the WSDL 1.2 specification does not provide as wide coverage for other schema languages, it does allow for their use within WSDL interfaces.

Message Elements

Once we have our types, we can move on to the business of specifying exactly how consumers can interact with the Web service. The message declarations compose the types that we have defined (and those that we are borrowing from other schemas) into the expected input, output and fault messages that the Web service will consume and produce. If we take our simple stock ticker Web service as an example, we can imagine a number of messages the Web service would be expected to exchange as shown in Figure 3-28.

```
<wsdl:message name="StockPriceRequestMessage">
  <wsdl:part name="symbol" element="stockQ:symbol"/>
</wsdl:message>
<wsdl:message name="StockPriceRespnseMessage">
  <wsdl:part name="price" element="stockQ:StockPriceType"/>
</wsdl:message>
<wsdl:message name="StockSymbolNotFoundMessage">
  <wsdl:part name="symbol" element="stockQ:symbol"/>
</wsdl:message>
```

Figure 3-28 The message elements.

As we see in Figure 3-28, a WSDL message declaration describes the (abstract) form of a message that a Web service sends or receives. Each message is constructed from a number of (XML Schema) typed part elements—which can come from the types part of the description or an external schema that has been imported into the same WSDL document—and each part is given a name to ease the insertion and extraction of particular information from a message. The name given to a part is unconstrained by WSDL but it is good practice to make the part name descriptive as one would when naming programming language variables.

In this example we have three possible messages: StockPriceRequestMessage, StockPriceResponseMessage, and StockSymbolNotFoundMessage, each of which carries some information having to do with stock prices and, because it is good practice to do so, whose name is indicative of its eventual use in the Web service.

PortType Elements

A portType defines a collection of operations within the WSDL document. Each operation within a portType combines input, output, and fault messages taken from a set of messages like those defined in Figure 3-28.

In the example shown in Figure 3-29, the StockBrokerQueryPortType declares an operation called GetStockPrice which is designed to allow users' systems to ask for the price of a particular equity.

The input to this operation is provided by a StockPriceRequestMessage message. The contents of this message are understood by the implementing Web service, which formu-

```
<wsdl:portType name="StockBrokerQueryPortType">
  <wsdl:operation name="GetStockPrice">
    <wsdl:input message="tns:StockPriceRequestMessage"/>
    <wsdl:output message="tns:StockPriceResponseMessage"/>
    <wsdl:fault name="UnknownSymbolFault"
      message="tns:StockSymbolNotFoundMessage"/>
</wsdl:portType>
```

Figure 3-29 Defining `portType` elements.

lates a response in an `output StockPriceRequestMessage` message that contains the details of the stock price for the equity requested.

Any exceptional behavior is returned to the caller through a fault called `UnknownSymbolFault` which is comprised from a `StockSymbolNotFoundMessage` message. Note that `portType` fault declarations have an additional name attribute compared to input and output messages, which is used to distinguish the individual faults from the set of possible faults that an operation can support.

Of course not all operations are so orthodox with a single input, output, and fault message, and so we have a variety of possible message exchange patterns described by the `operation` declarations within a `portType`, as follows:

- Input-Output: When the input message is sent to the service, either the output message is generated or one of the fault messages listed is generated instead.
- Input only: When a message is sent to the service, the service consumes it but does not produce any output message or fault. As such no output message or fault declarations are permitted in an operation of this type.
- Output-Input: The service generates the output message and in return the input message or one of the fault messages must be sent back.
- Output-only: The service will generate an output message, but does not expect anything in return. Fault messages are not allowed in this case.

Note that WSDL 1.2 changes the syntax of the `portType` declaration, renaming it `interface`. It also supports a useful new feature in the form of the `extends` attribute, which allows multiple `interface` declarations to be aggregated together and further extended to produce a completely new `interface`. For example, consider the situation where our simple stock Web service needs to evolve to support basic trading activities in addition to providing stock quotes. Using the `extends` mechanism, a new `interface` can be created which possesses all of the operations from each `interface` that it extends, plus any additional operations the developer chooses to add to it as exemplified in Figure 3-30.

The only potential pitfall when extending an `interface` is where names clash. For instance, an extending `interface` should take care not to call its operations by the same name

```
<wsdl:message name="BuyStockRequestMessage">
  <wsdl:part name="symbol" element="stockQ:symbol"/>
  <wsdl:part name="amount" element="xs:positiveInteger"/>
  <wsdl:part name="bid" element="stockQ:StockPriceType"/>
</wsdl:message>
<wsdl:message name="BuyStockResponseMessage">
  <wsdl:part name="symbol" element="stockQ:symbol"/>
  <wsdl:part name="amount" element="xs:positiveInteger"/>
  <wsdl:part name="price" element="stockQ:StockPriceType"/>
</wsdl:message>
<wsdl:message name="BidRejectedMessage">
  <wsdl:part name="symbol" element="stockQ:symbol"/>
  <wsdl:part name="amount" element="xs:positiveInteger"/>
  <wsdl:part name="bid" element="stockQ:StockPriceType"/>
  <wsdl:part name="asking" element="stockQ:StockPriceType"/>
</wsdl:message>

<wsdl:interface name="StockBrokerQueryPurchaseInterface"
  extends="tns:StockBrokerQueryInterface" >
  <wsdl:operation name="BuyStock">
    <wsdl:input message="tns:BuyStockRequestMessage"/>
    <wsdl:output message="tns:BuyStockRequestMessage"/>
    <wsdl:fault name="UnknownSymbolFault"
      message="tns:StockSymbolNotFoundMessage"/>
    <wsdl:fault name="BidRejectedFault"
      message="tns:BidRejectedMessage"/>
</wsdl:interface>
```

Figure 3-30 Extending `interface` definitions.

as operations from any `interface` that it extends unless the operations are equivalent. Furthermore, the designer of a new `interface` that extends multiple existing `interface` declarations must take care to see that there are no name clashes between any of the `interface` declarations as well as with the newly created `interface`.

Bindings

The `bindings` element draws together the `portType` and `operation` elements into a form suitable for exposing to the network. Bindings contain information that dictates how the format of the abstract messages is mapped onto the features of a particular network-level protocol.

While WSDL supports bindings for a number of protocols including HTTP GET and POST, and MIME, we are primarily interested in the SOAP binding for our simple stock quote `portType` from Figure 3-29, which is presented in Figure 3-31.

```
<wsdl:binding name="StockBrokerServiceSOAPBinding"
  type="tns:StockBrokerQueryPortType">
<soap:binding  styleDefault="document"
  transport="http://www.w3.org/2002/12/soap/bindings/HTTP/"
  encodingStyleDefault="http://stock.example.org/schema" />
  <wsdl:operation name="GetStockPrice">
    <soap:operation
      soapAction="http://stock.example.org/getStockPrice"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
    <wsdl:fault>
      <soap:fault name="StockSymbolNotFoundMessage"/>
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>
```

Figure 3-31 A SOAP binding.

The binding shown in Figure 3-31 binds the abstract `portType` defined in Figure 3-29 to the SOAP. It states how each of the message components of the operation defined in the `StockBrokerQueryPortType` is mapped onto its SOAP equivalent.

Starting from the top, we see a name for the binding (which is later used to tie a binding to a physical network endpoint) and the `portType` for which this binding is specified.

We then use elements from the WSDL SOAP binding specification to declare a binding for SOAP document-style exchanges, which is expressed as the default mode for this binding through the `styleDefault="document"` attribute. The encoding of the documents exchanged is defined by the stock broker schema `encodingStyleDefault="http://stock.example.org/schema"`. The fact that the service uses document-style SOAP and has its own schema means that it is a document-literal Web service.

Finall,y we see that the binding is for SOAP over the HTTP protocol as specified by the `transport="http://www.w3.org/2002/12/soap/bindings/HTTP/"` attribute. Each of these options is set as the default for the entire binding though both the style and encoding can be changed, if necessary, on a per-message basis.

This binding contains a single operation, namely `GetStockPrice`, which maps each of the input, output, and fault elements of the `GetStockPrice` operation from the `StockBrokerQueryPortType` to its SOAP on-the-wire format. The `soapAction` part of the operation binding is used to specify the HTTP SOAPAction header, which in turn can be used by SOAP servers as an indication of the action that should be taken by the receipt of the message at runtime—which usually captures the name of a method to invoke in a service implementation.

The soap:body elements for both the wsdl:input and wsdl:output elements provide information on how to extract or assemble the different messages inside the SOAP body. Since we have chosen literal encoding and document style for our messages (via the use="literal" and styleDefault="document" attribute), each part of a corresponding message is simply placed as a child of the soap:body element of the SOAP envelope. Had we been using RPC-style SOAP, then the direct child of the soap:body would be an element with the same name as the operation, with each message part as a child, as per SOAP RPC style, as contrasted with document style in Figure 3-32.[7]

```
<!-- RPC style -->
<soap:body>
  <GetStockPrice xmlns:gsp="http://stock.example.org/wsdl"
    xmlns:stockQ="http://stock.example.org/schema">
    <stockQ:symbol>MSFT</stockQ:symbol>
  </GetStockPrice>
</soap:body>

<!-- Document style -->
<soap:body>
  <stockQ:symbol
    xmlns:stockQ="http://stock.example.org/schema">
      MSFT
  </stockQ:symbol>
</soap:body>
```

Figure 3-32 Example SOAP RPC-style "Wrapping" element.

Note that the WS-I basic profile has mandated that only messages defined with element can be used to create document-oriented Web services, and messages defined with type cannot.

Of course, the value of SOAP is not only that it provides a platform-neutral messaging format, but the fact that the mechanism is extensible through headers. To be of use in describing SOAP headers, the WSDL SOAP binding has facilities for describing header content and behavior. For example, imagine that the query operation for which we have already designed a SOAP binding in Figure 3-31 evolves such that only registered users can access the service and must authenticate by providing some credentials in a SOAP header block as part of an invocation. The WSDL interface for the service obviously needs to advertise this fact to users' applications or no one will be able to access the service.

7. Note: this is not SOAP-encoded, just RPC-style (i.e., wrapped in an element that is named indicatively of the method that the message should be dispatched to).

The WSDL fragment shown in Figure 3-33 presents a hypothetical `soap:header` declaration within the `wsdl:input` element which mandates that a header matching the same namespace as the `userID` message (as declared earlier in the document) is present, and will be consumed by the ultimate receiver of the incoming SOAP message.

```
<wsdl:message name="UserID"
  targetNamespace="http://security.example.org/user">
  <wsdl:part name="signature" type="xs:string"/>
  <wsdl:part name="session" type="xs:anyURI"/>
</wsdl:message>

<wsdl:input>
  <soap:body use="literal"/>
  <soap:header use="literal" message="tns:UserIDMessage"/>
</wsdl:input>

<wsdl:output
  xmlns:sec="http://security.example.org/user">
  <soap:body use="literal"/>
  <soap:headerfault message="sec:UserID" part="signature"/>
</wsdl:output>
```

Figure 3-33 Describing SOAP headers.

Correspondingly, a `soap:headerfault` element is present in the `wsdl:output` element to report back on any faults that occurred while processing the incoming header. If a fault does occur while processing the header, this `soap:headerfault` element identifies the user's signature that caused the problem. This information, which amounts to a "user unknown" response, can then be used at the client end to perhaps prompt the end user to re-enter a pass phrase.

Note that an error such as an incorrect signature is propagated back through the header mechanism and not through the body, since the SOAP specification mandates that errors pertaining to headers must be reported likewise through header blocks.

Services

The services element finally binds the Web service to a specific network-addressable location. It takes the bindings declared previously and ties them to a `port`, which is a physical network endpoint to which clients bind over the specified protocol.

Figure 3-34 shows a service description for our stockbroker example. It declares a service called `StockBrokerService`, which it defines in terms of a port called `StockBrokerServiceSOAPPort`. The port is itself defined in terms of the `StockBrokerServiceSOAPBinding` binding, which we saw in Figure 3-31, and is exposed to the network at the address `http://stock.example.org/` to be made accessible through the endpoint specified at the `soap:address` element.

```
<wsdl:service name="StockBrokerService">
  <wsdl:port name="StockBrokerServiceSOAPPort"
   binding="tns:StockBrokerServiceSOAPBinding">
    <soap:address
        location="http://stock.example.org/"/>
  </wsdl:port>
</wsdl:service>
```

Figure 3-34 A `service` element declaration.

Managing WSDL Descriptions

While the service element is the final piece in the puzzle as far as an individual WSDL document goes, that's not quite the end of the story. For simple one-off Web services, we may choose to have a single WSDL document that combines both concrete and abstract parts of the interface. However, for more complex deployments we may choose to split the abstract parts into a separate file, and join that with a number of different concrete bindings and services to better suit the access pattern for those services.

For example, it may be the case that a single abstract definition (`message`, `portType`, and `operation` declarations) might need to be exposed to the network via a number of protocols, not just SOAP. It might also be the case that a single protocol endpoint might need to be replicated for quality of service reasons or perhaps even several different organizations each want to expose the same service as part of their Web service offerings. By using the WSDL import mechanism, the same abstract definition of the service functionality can be used across all of these Web services irrespective of the underlying protocol or addressing. This is shown in Figure 3-35 where MIME, HTTP, and SOAP endpoints all share the same abstract functionality yet expose that functionality to the network each in their own way. Additionally, the SOAP protocol binding has been deployed at multiple endpoints which can be within a single administrative domain or spread around the whole Internet and yet each service, by dint of the fact that they share the same abstract definitions, is equivalent.

If a WSDL description needs to include features from another WSDL description or an external XML Schema file, then the `import` mechanism is used. It behaves in a similar fashion to the XML Schema `include` feature where it can be used to include components from other WSDL descriptions. We have already seen how the WSDL import mechanism is used in Figure 3-27 where the XML Schema types from the stockbroker schema were exposed to the stock broking WSDL description, as follows:

```
<wsdl:import namespace="http://stock.example.org/schema"
  location="http://stock.example.org/schema"/>
```

The `import` feature of WSDL means that a WSDL description can leverage existing XML infrastructure—previously defined schemas for in-house documents, database schemas, existing Web services, and the like—without having to reproduce those definitions as part of its own description.

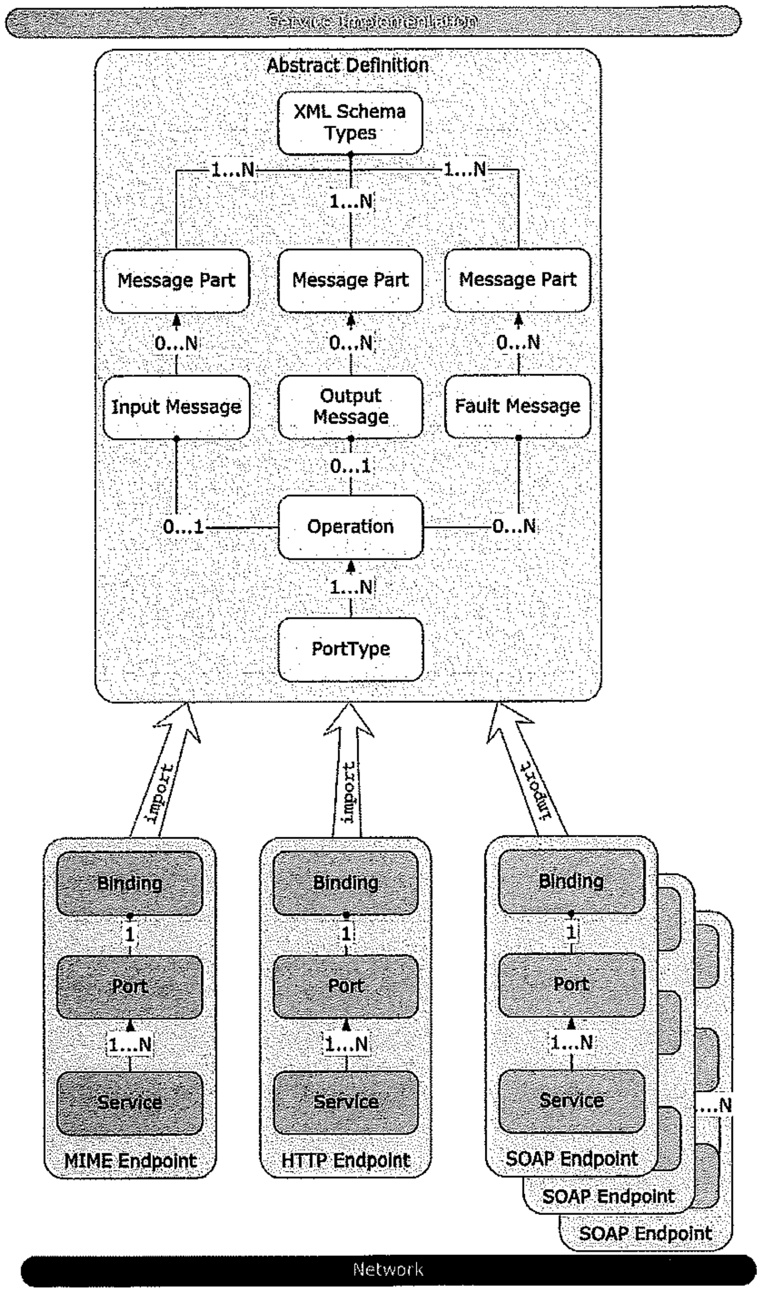
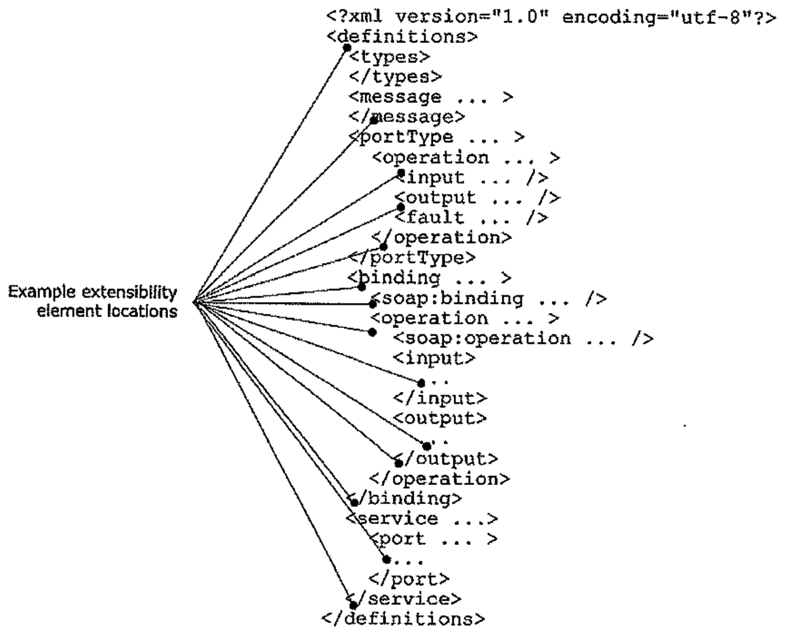


Figure 3-35 Including abstract WSDL descriptions for concrete endpoints.

Extending WSDL[8]

As Web services technology has advanced and matured, WSDL has begun to form the basis of higher-level protocols that leverage the basic building blocks that it provides, to avoid duplication of effort. Many of the technologies that we are going to examine throughout this book extend WSDL via such means to their own purpose. However, where SOAP offers header blocks as its extensibility mechanism for higher-level protocols to use, WSDL offers extension elements based on the XML Schema notion of substitution groups (see Chapter 2).

In the WSDL schema, several (abstract) global element declarations serve as the heads of substitution groups. In addition, the WSDL schema defines a base type for use by extensibility elements as a helper to ensure that the necessary substitution groups are present in any extensions. While it is outside the scope of this book to present the WSDL schema in full, there exists in the schema extensibility elements which user-defined elements can use to place themselves at *any* point within a WSDL `definition`. There are extensibility elements that allow extensions to appear at global scope, within a `service` declaration, before the `port` declaration, in a `message` element before any `part` declarations and any other point in a WSDL description, as shown in Figure 3-36.



```
<?xml version="1.0" encoding="utf-8"?>
<definitions>
  <types>
  </types>
  <message ... >
  </message>
  <portType ... >
    <operation ... >
      <input ... />
      <output ... />
      <fault ... />
    </operation>
  </portType>
  <binding ... >
    <soap:binding ... />
    <operation ... >
      <soap:operation ... />
      <input>
        ...
      </input>
      <output>
        ...
      </output>
    </operation>
  </binding>
  <service ...>
    <port ... >
      ...
    </port>
  </service>
</definitions>
```

Figure 3-36 WSDL substitution group heads.

8. This section based on a draft version of the WSDL 1.2 specification.

For example, the `soap` elements that we have seen throughout the bindings section of our WDSL description are extensibility elements. In the schema for those elements, they have been declared as being part of the substitution group `bindingExt` which allows them to legally appear as part of the WSDL `bindings` section.

Additionally, third-party WSDL extensions may declare themselves as mandatory with the inclusion of a `wsdl:required` attribute in their definitions. Once a `required` attribute is set, any and all validation against an extended WSDL document must include the presence of the corresponding element as a part of the validation.

Extensibility elements are commonly used to specify some technology-specific binding. They allow innovation in the area of network and message protocols without having to revise the base WSDL specification. WSDL recommends that specifications defining such protocols also define any necessary WSDL extensions used to describe those protocols or formats.[9]

Using SOAP and WSDL

While many of the more advanced features of the emerging Web services architecture are still being built into many of the platforms, support for SOAP and WSDL in most vendors' Web services toolkits is widespread and makes binding to and using Web services straightforward. In this section, we investigate how a typical application server and can be used to deploy our simple banking example, and how it can be later consumed by a client application. The overall architecture can be seen in Figure 3-37.

The architecture for this sample is typical of Web services applications that routinely combine a variety of platforms. In Figure 3-37, we use Microsoft's .Net and Internet Information Server to host the service implementation, but we use the Java platform and the Apache AXIS Web service toolkit to consume this service and drive the application.

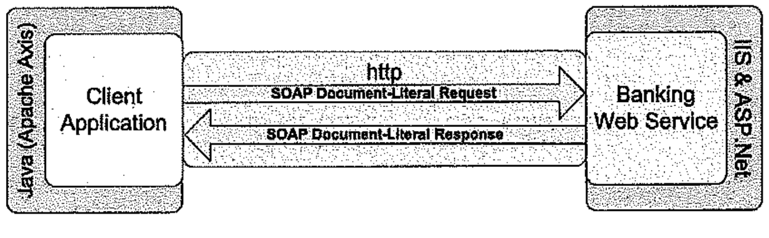


Figure 3-37 Cross-platform banking Web service example.

9. From WSDL 1.2 specification, http://www.w3.org/TR/wsdl12/.

Service Implementation and Deployment

The implementation of our banking service is a straightforward C# class, and is shown in Figure 3-38.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Web;
using System.Web.Services;

[WebService(Namespace="http://bank.example.org")]
public class BankService : System.Web.Services.WebService
{
  [WebMethod]
  public string openAccount(string title,
                            string surname,
                            string firstname,
                            string postcode,
                            string telephone)
  {
    BankEndSystem bes = new BackEndSystem();
    string accountNumber = bes.processApplication(title,
                                                   surname,
                                                   firstname,
                                                   postcode,
                                                   telephone);
    return accountNumber;
  }
}
```

Figure 3-38 A simple bank Web service implementation.

Most of the work for this service is done by some back-end banking system, to which our service delegates the workload. Our service implementation just acts as a kind of gateway between the Web service network to which it exposes our back-end business logic, and the back-end systems themselves to which it delegates work it receives from Web services clients. This pattern is commonplace when exposing existing systems via Web services, and makes good architectural sense since the existing system does not have to be altered just to add in Web service support.

The key to building a successful Web service, even one as simple as our bank account example, is to ensure that the orthogonal issues of service functionality and deployment are kept separate. That is, do not allow the implementation of your system to change purely because you intend to expose its functionality as a Web service.

It is a useful paradigm to treat your Web services as "user interfaces" through which users (in most cases other computer systems) will interact with your business systems. In the same way that you would not dream of putting business rules or data into human user interfaces, then you should not place business rules or data into your Web service implementations. Similarly, you would not expect that a back-end business system would be updated simply to accommodate a user interface, and you should assume that such mission-critical systems should not be altered to accommodate a Web service deployment.

When deployed into our Web services platform (in this example, Microsoft's IIS with ASP.Net), the associated WSDL description of the service is generated by inspection of the implementation's interface and made available to the Web. The resultant WSDL[10] is shown in Figure 3-39.

It is important to bear in mind, that although the WSDL shown in Figure 3-39 is intricate and lengthy for a simple service, the effort required to build it is practically zero because of tool support. The only issue that this should raise in the developer's mind is that their chosen platform and tools should handle this kind of work on their behalf. WSDL should only be handcrafted where there a specific need to do something intricate and unusual that tool support would not facilitate.

Binding to and Invoking Web Services

Once the service has been deployed and its endpoint known by consumers, clients can bind to it by using their client-side Web services toolkits to create proxies. A proxy is a piece of code that sits between the client application and the network and deals with all of the business of serializing and deserializing variables from the client's program into a form suitable for network transmission and back again. The client application, therefore, never has to be aware of any network activity and is simpler to build.

10. The WDSL description generated by ASP.Net is richer than that shown here since it also includes HTTP GET and HTTP POST bindings. However, we are predominantly interested in SOAP as the Web services transport, and so the HTTP bindings have been removed.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:bank="http://bank.example.org"
  targetNamespace="http://bank.example.org"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xs:schema elementFormDefault="qualified"
       targetNamespace="http://bank.example.org">
      <xs:element name="openAccount">
        <xs:complexType>
          <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="1"
               name="title" type="xs:string"/>
            <xs:element minOccurs="0" maxOccurs="1"
               name="surname" type="xs:string"/>
            <xs:element minOccurs="0" maxOccurs="1"
               name="firstname" type="xs:string"/>
            <xs:element minOccurs="0" maxOccurs="1"
               name="postcode" type="xs:string"/>
            <xs:element minOccurs="0" maxOccurs="1"
               name="telephone" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="openAccountResponse">
        <xs:complexType>
          <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="1"
               name="openAccountResult" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="string" nillable="true"
         type="xs:string"/>
    </xs:schema>
  </types>
  <message name="openAccountSoapIn">
    <part name="parameters" element="bank:openAccount"/>
  </message>
  <message name="openAccountSoapOut">
    <part name="parameters"
     element="bank:openAccountResponse"/>
  </message>
  <portType name="BankServiceSoap">
```

Figure 3-39 Bank service auto-generated WSDL description.

```
    <operation name="openAccount">
      <input message="bank:openAccountSoapIn"/>
      <output message="bank:openAccountSoapOut"/>
    </operation>
  </portType>
  <binding name="BankServiceSoap"
   type="bank:BankServiceSoap">
    <soap:binding
         transport="http://schemas.xmlsoap.org/soap/http"
         style="document"/>
    <operation name="openAccount">
      <soap:operation
           soapAction="http://bank.example.org/openAccount"
           style="document"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="BankService">
    <port name="BankServiceSoap"
     binding="bank:BankServiceSoap">
      <soap:address
          location="http://localhost/dnws/BankService.asmx"/>
    </port>
  </service>
</definitions>
```

Figure 3-39 Bank service auto-generated WSDL description (continued).

In our example, the serialization and deserialization is to SOAP from Java and back again, and is handled by a proxy generated by the Apache AXIS `WSDL2Java` tool. This tool parses WSDL at a given location and generates a proxy class which allows client code to communicate with that service. For example, the proxy code generated by this tool when it consumed our bank example service is shown in Figure 3-40.

```
/**
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package org.example.bank;
import java.lang.String;

public class BankServiceSoapStub
                extends org.apache.axis.client.Stub
                implements org.example.bank.BankServiceSoap {

  // Data members removed for brevity

  public BankServiceSoapStub()
                          throws org.apache.axis.AxisFault {
     this(null);
  }

  // Other constructors removed for brevity

  private org.apache.axis.client.Call createCall()
                            throws java.rmi.RemoteException {
       // Implementation removed for brevity
       return _call;
    }
    catch (java.lang.Throwable t) {
       throw new org.apache.axis.AxisFault("Failure trying" +
                              " to get the Call object", t);
    }
  }

  public String openAccount(String title, String surname,
                            String firstname,
                            String postcode,
                            String telephone)
                            throws java.rmi.RemoteException {
     // Implementation removed for brevity
  }
}
```

Figure 3-40 Apache AXIS auto-generated proxy for the bank Web service.

The proxy class shown in Figure 3-40 allows the client of the Web service to call its functionality with a call as simple as the likes of:

```
bankAccountService.openAccount("Mr", "Aneurin", "Bevan",
                               "ABC 123", "0207 123 4567")
```

without having to worry about the fact that on the wire, the proxy has sent a SOAP message that looks like that shown in Figure 3-41 below:

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <openAccount xmlns="http://bank.example.org">
      <title>Mr</title>
      <surname>Bevan</surname>
      <firstname>Aneurin</firstname>
      <postcode>ABC 123</postcode>
      <telephone>0207 123 4567</telephone>
    </openAccount>
  </soap:Body>
</soap:Envelope>
```

Figure 3-41 Proxy generated SOAP message.

At the receiving end, the bank service's SOAP server will retrieve this SOAP from the network and turn it into something meaningful (in our case C# objects) before passing it to the service implementation. The service implementation grinds away at its task, producing some result in its own proprietary format before passing it back to the underlying Web services platform to serialize its results into the appropriate network format (i.e., a SOAP message) and return it to the caller. At this point the service invocation has finished and the resources used during the execution of that service can be freed.

Where's the Hard Work?

For simple interactions, there isn't any hard work for the developer to do because SOAP toolkits are sufficiently advanced enough to automate this. For example, we didn't have to worry about the style of SOAP encoding or how the marshalling occurred in any of our bank account examples, even though we crossed networks, servers, and even languages and platforms.

Though it may seem from these examples that Web services is an automation utopia, it is not. While it is true that for the majority of cases, simple interactions can be automated (though auto-generation of WSDL from service implementation code and auto-generation of proxies from WSDL descriptions), this is about as far as toolkits have advanced.

Given that this book extends beyond this third chapter, it is safe to assume that we're going to have to roll up our shirt sleeves at some point and patch the gaps that the current set of Web services toolkits inevitably leaves. It is in these subsequent chapters where we will find the hard work!

Summary

SOAP is the protocol that Web services use to communicate. It is an XML-based protocol that specifies a container called an Envelope, which stores application payload in a second container, called the Body, and additional (usually contextual) information inside a third container called the Header. The SOAP specification describes a processing model where application messages (and their associated headers) can pass through intermediary processing nodes between the sender and receiver, where the information stored in the SOAP header blocks can be used by those intermediaries to provide various quality of service characteristics. For example, the headers may contain routing information, transaction context, security credentials, or any other protocol information.

WSDL is an interface description language for Web services and like SOAP, WSDL is currently popularized by its 1.1 version, which is due to be superseded by WSDL 1.2. A WSDL interface is composed from a number of elements, each building on the previous, from simple type and message declarations, culminating in a network addressable entity which uses the defined types and messages to expose operations onto the Web.

Though SOAP and WSDL are undoubtedly important protocols in their own right, when drawn together through tool support, their potential is significantly enhanced. Web services toolkits can consume the WSDL offered by a service and automatically generate the code to deal with messages in the format that the service expects, while providing a straightforward API to the developer.

Architect's Note

- SOAP 1.1 is the most widely adopted version of the SOAP specification. However, SOAP 1.2 has now reached W3C recommendation status and thus SOAP 1.1 is now considered deprecated.
- SOAP RPC is quick and easy, but may lead to applications with too tight a level of coupling. Exchanging larger documents is preferable, even if it means writing handler code to deal with them.
- XML-Native applications should not use SOAP-RPC; they should use the XML vocabularies that they have already developed, and use those vocabularies as the basis of their communication via document-oriented SOAP.
- Be prepared for a shift in the Web services architecture, and ensure your services can support "Web-friendly" access where appropriate.
- Do not deploy a Web service without its WSDL description—a service is naked without it.
- Use tool support—it is wasted effort to do for yourself what a tool can do more easily, more quickly, and more accurately.

CHAPTER 4

UDDI—Universal Description, Discovery, and Integration

When UDDI came on the scene, its champions positioned the new technology as the savior of e-business. Businesses along a value chain would use UDDI registries to dynamically and automatically select new business partners, locate the electronic services implemented by those partners and start executing e-commerce transactions with them. This would revolutionize how businesses operate: wipe out the need for human interaction in many business tasks, reduce overheads and middleman costs, and fundamentally enable a dynamic and fluid e-business environment.

Today, it is difficult to find companies that are truly using UDDI, and UDDI registries boast a relatively small number of entries. Does this mean that UDDI is DOA (dead on arrival)? By just looking at the list of some of the companies that are backing the UDDI project, one would conclude probably not.

So, how will UDDI pan out? What will enterprises do with UDDI? What do enterprise architects have to know about UDDI? In this chapter, we delve into these issues and take a practical approach to UDDI and its fit within the enterprise Web services picture. We look at the latest release of the UDDI specification—Version 3—and take a closer look at some of the key architectural changes.

UDDI at a Glance

The UDDI is a registry and a protocol for publishing and discovering Web services. As Web services are a standards-based, open, and platform-independent means of accessing the functional capabilities of other companies, UDDI is the associated standards-based, open, and platform-independent means of publishing and locating these services. The latest information about UDDI and the UDDI community can be found at http://www.uddi.org.

As more and more companies start driving toward a services-oriented architecture, and Web services in particular, for their enterprise application infrastructure, the issue of *locating* Web services becomes increasingly important. When companies initially began experimenting with Web services behind the firewall, there was no question of locating or discovering services as each company controlled everything—both the services and the consuming applications.

As these experimental applications were migrated across the firewall, the services they consumed were augmented to include Web services from a handful of partner companies. All of the necessary information about these services was known *a priori*, and still the need to discover services was unnecessary.

As these applications were further scaled, there emerged a need to answer questions such as: Which business partners have this service? What types of services do these partners offer? As more business partners adopted Web services, the process of obtaining these answers became difficult, not to mention time consuming. The old methods of jointly agreeing on services and their interfaces were no longer feasible. Neither was manually calling up business partners to get a list of their latest service offerings.

There emerged a need for a registry where service providers could publish not only a list of their services but also information necessary to use the services. At the same time, businesses could search through the registry to discover these service providers and their services. These are the underpinnings of UDDI.

Analogies with Telephone Directories

UDDI shares some striking similarities with telephone directories (e.g., yellow pages). As such, the analogy is an effective vehicle for describing the capabilities and usefulness of UDDI.

A phone book allows people to search for other people and businesses, get their contact information, and then directly contact the person or business. Phone books allow various modes of searching, whether it be an alphabetical listing of people or business names (as in the white pages) or through categories of businesses.

Anyone can view the listings of a phone directory; in fact, the more people who view and use the phone book, the more valuable it is. However, only the phone company or its authorized agent publishes the phone book. When adding or updating entries, the requester must validate his or her identity and provide evidence that he or she has the right to add or change the information.

The importance of phone books grows as the need to locate more people and businesses increases. When there are just a handful of people and businesses and few new additions, phone books are not as important. It is easy to keep track of contact information, or gather the information when necessary. However, as the base of people and businesses becomes large and there are continuous changes—both in people and businesses being added or removed from the listings or their contact information changes—phone books become critical. They provide a centralized source for contact information.

UDDI is quite similar. Instead of a directory of telephone numbers, UDDI is a directory of Web services that are available from different vendors. UDDI provides a means of adding new services, removing existing services, and changing the contact (i.e., endpoint) information for services.

Most UDDI implementations also have some of the same constraints as phone books. Only authenticated users (e.g., service providers) can add or change their information on the UDDI registry. Non-authenticated users are not allowed to change any information on a UDDI registry, and only authenticated users can change their own information. This policy prevents maliciously motivated changes to UDDI entries. Any user can access a UDDI registry for read-only purposes.

Both telephone directories and UDDI registries provide a means to locate a vendor or provider of a particular service. For telephone directories, contact information is basically a phone number and perhaps may also include an address. Contact information in a UDDI registry consists of information about the service provider as well as technical information about the Web service itself. Conceptually, the information available in an UDDI registry is similar to that in the white, green, and yellow pages of the phone book. In UDDI, the segmentation of information that is available and searchable can be thought of as follows:

- White Pages: Contact information about the service provider company. This information includes the business or entity name, address, contact information, other short descriptive information about the service provider, and unique identifiers with which to facilitate locating this business.
- Yellow Pages: Categories (taxonomies) under which Web services implementing functionalities within those categories can be found.
- Green Pages: Technical information about the capabilities and behavioral grouping of Web services.

How are people supposed to use an UDDI registry? First, let's look at how people use telephone books. When using the phone book to contact a business, the user has a product or service in mind. From her past purchases, she may also have a few businesses in mind that sell that product. The user looks up these business names to find their contact information. Otherwise, the user searches through product categories to locate a vendor. Once she has identified a suitable vendor, she looks up the corresponding phone number and contacts the vendor.

What if there are multiple possible vendors? How does a user determine the winner? The winning vendor may be chosen based on price. The user may prefer to do business with a particular vendor if she has done a lot of business with the vendor in the past. The user may shy away from a vendor because the vendor has been unreliable or has delivered shoddy product.

Using a UDDI registry is similar to using a standard telephone directory. Users will search through the UDDI registry for an appropriate Web service that meets their needs. The search may involve a straightforward name lookup, or may involve searching through the taxonomies (service provider categories) provided by the UDDI registry. What do you do when there are multiple Web services that may potentially meet your needs? You have to pick a winner based on whatever metrics are important to you. These may be cost, personal preferences, or other business relationships.

Figure 4-1 depicts the similarities between telephone directory books and UDDI registries.

Although there are strong similarities between these, there are some places where the analogy breaks down. First, each Web service implements a unique API. Although this is not by specification, it is statistically unlikely that two independent programmers will define and implement the same programmatic interface. Unlike different phone numbers that merely provide unique identification or routing information for phone calls, different Web service APIs are more analogous to using a different and unique phone number for communicating with each person or business.

Second, people will not typically interact directly with UDDI registries as they do with phone books. This is because the information available on UDDI is not people-friendly. Instead, portals and software tools facilitate access to UDDI registries. Many of the same middleware and application development tools that support Web service development allow users to easily add new services to the UDDI registry. These and other tools also allow browsing through the services on UDDI, and many augment the information available on UDDI with their own analysis. This analysis may include quality-of-service information and additional information helpful in using the Web services.

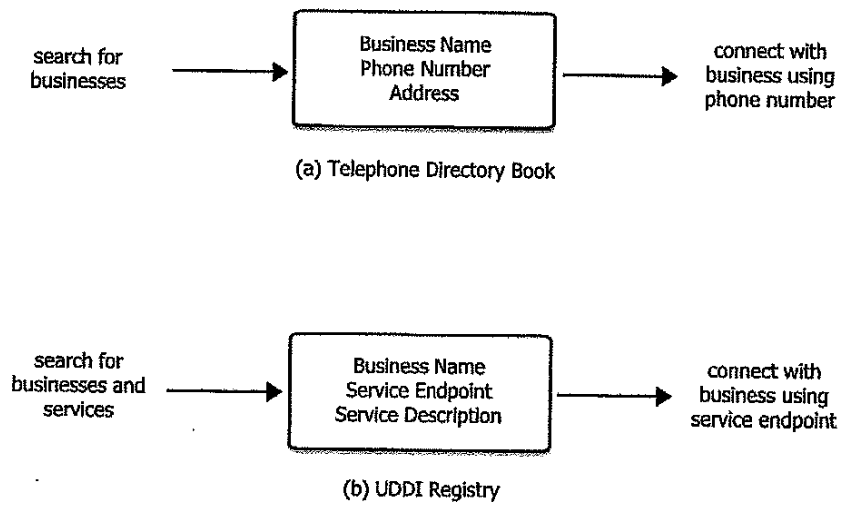

search for businesses
Business Name
Phone Number
Address
connect with business using phone number
(a) Telephone Directory Book
search for businesses and services
Business Name
Service Endpoint
Service Description
connect with business using service endpoint
(b) UDDI Registry

Figure 4-1 Similarities between (a) telephone directory books and (b) UDDI registries.

Another key difference is that within organizations, UDDI will probably be accessed by two different groups of people. Unlike phone books, interactions with UDDI require an understanding of more issues. For example, which Web service to use for a particular application is not only based on technical needs and QoS requirements, other strategic and business issues also come into the mix. There may be existing relationships between two companies that require the use of a particular company's Web service over that of another. Or, it may make strategic sense for a company to use a particular Web service, even if other technically superior Web services exist. As such, a unique interaction of business issues together with technical issues comes together to determine which Web service to use for a particular application. Since most technical programmers are usually not party to such information, business analysts with an understanding of strategic business issues typically will select Web services by searching through UDDI registries and other related information portals. A programmer will then search the UDDI registry for that particular Web service's API, and implement the communications between the application and that Web service, as depicted in Figure 4-2.

A critical point to remember is that business issues are quite fluid. The dynamics of most business environments result in rapidly changing relationships. This, in turn, results in continuously changing or at least evolving business-driven requirements. Flexibility in selecting and consuming Web services is important.

It is a common misconception that applications can themselves dynamically select and consume Web services. Although one day software may become sufficiently smart to do this,

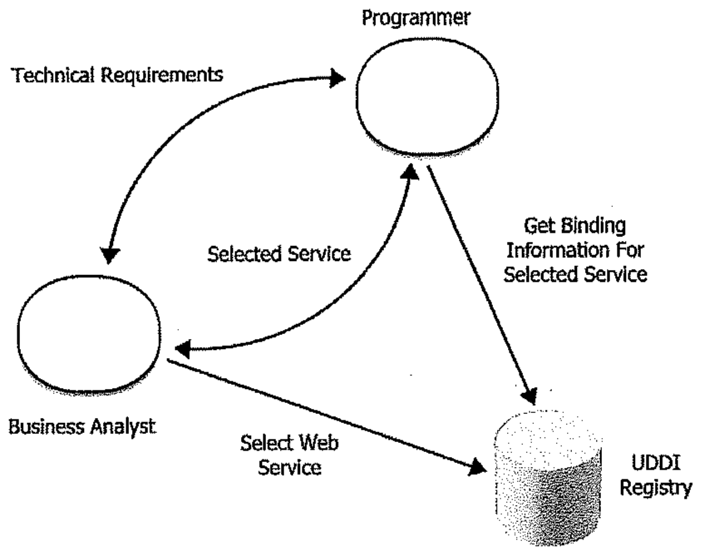


Figure 4-2 The typical roles played while interacting with an UDDI registry.

today selecting and consuming Web services requires some degree of human intervention. Some simple cases of automation do certainly exist, but automating the process in a general sense is not available today. Why not? Because each Web service implements a unique API that requires programmatic and perhaps architectural changes to the consuming application. Moreover, automating the process of selecting the appropriate Web service to consume is difficult and dynamic. Some newer tools support the use of business rules to automate (at a higher level) the process of service selection. Nonetheless, some level of human intervention is necessary.

The UDDI Business Registry

The UDDI Business Registry (UBR) is a global implementation of the UDDI specification. The UBR is a single registry for Web services. A group of companies operate and host UBR nodes, each of which is an identical copy of all other nodes. New entries or updates are entered into a single node, but are propagated to all other nodes.

The UBR is a key element of the deployment of Web services and provides the following capabilities:

- A centralized registration facility at which to publish and make others aware of the Web services a company makes available.
- A centralized search facility at which companies that require a particular service can locate businesses that provide that service as well as relevant information about that service.

A small group of companies operate and manage a set of UBR nodes. In July 2002, the UBR was updated to support version 2 of the UDDI specification. Initially, IBM, Microsoft, and SAP comprised the UBR V2, operating 3 UBR nodes. NTT Communications later launched an UBR node to become the fourth UBR V2 node. More than 10,000 businesses are registered with the initial three UBR nodes, publishing over 7,000 Web services. NTT expects to add another 1,000 businesses within the first operational year of the fourth UBR node.

Each UBR node provides a Web home page for human-friendly navigation of the registry as well as information about the use of the registry. Today, most searches for available Web services are done through human-friendly means: phone conversations between existing business partners, the home pages of the UBR, Web service aggregator portals such as www.xmethods.com, or standard Web search engines such as Google. UBR node home pages also provide other information pertaining to UDDI or to that particular UBR node. This information includes policies on data replication, publishing restrictions, and other administrative or usage issues.

UBR nodes also implement a simple API for direct electronic (computer-to-computer) access to the contents of the registry. The two most important and relevant features of the APIs are inquiry and publication.

The inquiry API allows searching through the registry for information about businesses, the Web services the business makes available, as well as implementation and interface information for each service.

The publication API allows adding, changing, and deleting business and service information within the registry.

Figure 4-3 depicts some typical means of accessing and interacting with an UDDI registry.

The URL access endpoint information of the home page, inquiry API, and publication API of each UBR node is different, and the information for each of the UBR V2 nodes is listed in Table 4-1. The publication API endpoint requires authentication and uses the HTTPS protocol, while the inquiry API and home page use standard HTTP.

The UBR operators also provide fully functional test environments where companies can develop and test their offerings without affecting other users. Some of these test nodes do not support version 2 of the UDDI specification as yet. Table 4-2 lists the endpoint access information for the test nodes of the UBR.

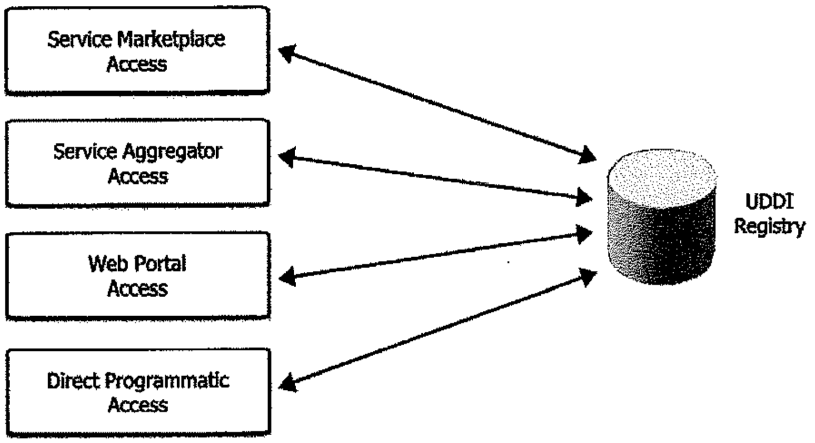

Service Marketplace Access
Service Aggregator Access
Web Portal Access
Direct Programmatic Access
UDDI Registry

Figure 4-3 The various means of accessing an UDDI registry.

Table 4-1 The operator node URLs for the UDDI Business Registry (UBR).

UBR Operator Node		URL
IBM	Home Page	http://uddi.ibm.com/
	Inquire API	http://uddi.ibm.com/ubr/inquireapi
	Publish API	https://uddi.ibm.com/ubr/publishapi
Microsoft	Home Page	http://uddi.microsoft.com/
	Inquire API	http://uddi.microsoft.com/inquire
	Publish API	https://uddi.microsoft.com/publish
SAP	Home Page	http://uddi.sap.com/
	Inquire API	http://uddi.sap.com/uddi/api/inquiry
	Publish API	https://uddi.sap.com/uddi/api/publish
NTT Com	Home Page	http://www.ntt.com/uddi/
	Inquire API	http://www.uddi.ne.jp/ubr/inquiryapi
	Publish API	https://www.uddi.ne.jp/ubr/publishapi

Table 4-2 The test node URLs for the UDDI Business Registry (UBR).

UBR Test Operator Node		URL
IBM	Home Page	http://uddi.ibm.com/testregistry/registry.html
	Inquire API	http://uddi.ibm.com/testregistry/inquiryapi
	Publish API	https://uddi.ibm.com/testregistry/publishapi
Microsoft	Home Page	http://test.uddi.microsoft.com/
	Inquire API	http://test.uddi.microsoft.com/inquire
	Publish API	https://test.uddi.microsoft.com/publish
SAP	Home Page	http://udditest.sap.com/
	Inquire API	http://udditest.sap.com/UDDI/api/inquiry
	Publish API	https://udditest.sap.com/UDDI/api/publish

Later in the chapter we look at how to programmatically access the information at these UBR nodes to locate the latest information about a particular Web service.

UDDI Under the Covers

In the remainder of this chapter, we discuss how to add entries to a UDDI registry as well as how to search for available services and build applications that consume those services. We will also briefly touch on the major sections of the UDDI specification.

The UDDI Specification

Version 3 is the most recent incarnation of the UDDI specification. Version 3 builds on and expands the foundations laid by versions 1 and 2 of the UDDI specification, and presents a blueprint for flexible and interoperable Web services registries. Version 3 also includes a rich set of enhancements as well as additional features, including improved security and new APIs. The entire UDDI specification can be found at http://www.uddi.org.

The major documents of the UDDI Version 3 specification are listed in Table 4-3.

Table 4-3 The major documents of the UDDI Specification version 3.

UDDI Version 3 Specification Documents	Synopsis
Features List	Brief overview of the key features in version 3.
Specification	The actual specification document.
XML Schemas	A set of XML Schema files that formally describe UDDI data structures.
WSDL Service Interface Descriptions	A set of files that describe the UDDI Version 3 WSDL interface definitions.

Unlike in previous versions, UDDI Version 3 consolidates the entire specification into a single document entitled the UDDI Version 3 Published Specification. This single document contains everything related to UDDI, and also contains all information necessary for developing a UDDI node, the Web services that are called by a UDDI node, or a client application that directly interacts with a UDDI registry.

UDDI Core Data Structures

Information representation within UDDI consists of instances of persistent data structures that are expressed in XML. It is these data structures that are persistently stored and managed by UDDI nodes. The UDDI specification refers to these as entities, and defines four core entity types as listed in Table 4-4.

Table 4-4 The different entity types defined by the UDDI Information Model.

Entity Type Name	Description
businessEntity	A business that provides a Web service.
businessService	A collection of related services offered by a business.
bindingTemplate	Technical information about a particular Web service.
tModel	Technical model information about a Web service that is used to determine whether a service is compatible with the client's needs.

Whether you intend to programmatically connect to a UDDI registry or manually browse through one, it is necessary to understand these core data structures. Central to the purpose of UDDI is the representation of information about Web services so they can be easily registered and classified by publishers as well as searched and consumed by client applications. As such the data structures used by UDDI provide not only technical interface information about a service itself, but also information necessary to classify, manage, and locate services. Figure 4-4 depicts the interrelationships between the core UDDI data structures.

The `businessEntity` entity type represents information about service providers within UDDI. This information includes detailed data about the name of the provider, contact information, and some other short descriptions of the provider. This information may also be

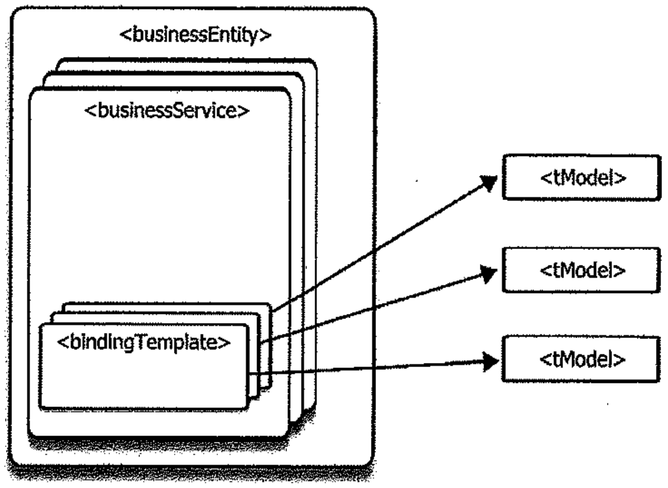


Figure 4-4 The interrelationship between the UDDI core data structures.

provided in multiple languages. The businessEntity structure does not necessarily have to refer to a business, but to any type of service provider, such as a department within an organization or a group.

One or more of the businessService entity types are contained within a businessEntity structure and represents information about the services offered by that businessEntity. The businessService entity type does not provide implementation or technical details, but instead is a logical grouping of Web services and provides information about the bundled purpose of a set of contained Web services.

One or more of the bindingTemplate entity types are contained within a businessService structure and provides technical information about a particular Web service. The bindingTemplate structure directly or indirectly provides descriptive technical information about an instance of a Web service, and includes a network location or endpoint of the service. The network location (access point) is usually a URL, but can be other network access points such as email addresses. The bindingTemplate structure also includes information about the type of Web service located at that access point through references to tModel entities as well as other parameters.

tModels, which are short for technical models, provide more detailed information about a Web service. tModels are reusable entities that are referenced from bindingTemplate structures and denote compliance with a shared concept or design. tModels are not contained within bindingTemplates, but instead are referenced. Distinct tModels exist for different interfaces and contracts that a Web service can comply with including specifications, transports, protocols, and namespaces. The set of tModels that a bindingTemplate refers to makes up a Web service's technical fingerprint. The actual documents and information identified by a tModel are not located within the UDDI registry itself, but instead the tModel provides pointers to the location where such documents can be found.

Two more UDDI entity types that are important are subscription and publisherAssertion. The subscription entity type describes the request to keep track of the evolution or changes to particular entities. The publisherAssertion entity type describes the relationship between one businessEntity and another businessEntity. There are many instances where multiple divisions within a large organization or a group of organizations want to make the relationship between them known in order to facilitate discovery of the services they provide. The individual divisions or organizations each have their own businessEntity, and the entity type publisherAssertion describes the relationship between two businessEntity structures. It is important to note that two organizations must assert the same relationship through the publisherAssertion for that relationship to be publicly available. This disallows the situation where one organization claims a relationship with another where in fact there is none.

Accessing UDDI

UDDI is itself a Web service and as such, applications can communicate with an UDDI registry by sending and receiving XML messages. This makes the access both language and platform independent.

Although it's possible, it is unlikely that programmers will deal with the low-level details of sending and receiving XML messages. Instead, client-side packages for different languages and platforms will emerge that facilitate programmatic access to UDDI.

Two such packages are UDDI4J and Microsoft's UDDI SDK, which are client-side APIs for communicating with UDDI from Java and .Net programs, respectively. UDDI4J was originally developed by IBM and released in early 2001 as an open source initiative. Later, HP joined and contributed to the initiative, developing much of the version 2 release. With the support of IBM and HP (as well as others), UDDI4J has become the de facto standard Java package for communicating with UDDI registries. More information about UDDI4J, including the latest releases and download bundles, can be found at the UDDI4J Project Web site at http://www-124.ibm.com/developerworks/oss/uddi4j/.

Figure 4-5 shows how UDDI4J facilitates programmatic access to an UDDI registry. With UDDI4J, programmers don't have to concern themselves with either the UDDI API or with forming and parsing XML messages. Instead, a new Java object, `UDDIProxy`, is instantiated to act as a proxy and represent the actual UDDI registry. Using setter methods, the proxy object is configured with the URLs of the actual registry location, as well as optional transport information. Essentially, using UDDI4J and just a few, simple lines of code, a Java application can open a communications channel to any UDDI registry.

```
// Create a new UDDIProxy object to connect to a registry
UDDIProxy proxy = new UDDIProxy ();

// Set the inquiry and publish URLs
proxy.setInquireURL (INQUIRE_URL);
proxy.setPublishURL (PUBLISH_URL);
```

Figure 4-5 Opening a connection to an UDDI registry using UDDI4J.

Once we've created the proxy object and set its inquire and publish URLs to the desired UDDI registry locations, we can use the methods that are defined for the `UDDIProxy` object to access and set various elements within the registry. Usually, programmers will use the `find_business`, `find_service`, and `find_tModel` methods to locate service providers, services, and `tModels`, respectively, based on search criteria, such as name (including partial names with wildcards) and categories.

Figure 4-6 shows a complete application using UDDI4J to connect to Microsoft's UDDI Business Registry (UBR) inquiry node and locate service providers whose name includes the string "abc". After an `UDDIProxy` proxy object for Microsoft's UBR inquiry node is set up, the `find_business` method is invoked to search for available business names that contain the substring "abc". The wildcard character '%' is used to specify that the substring may occur anywhere in the business name. Qualifiers, such as case-sensitive string matching, could have been added to the `find_business` method to further limit the search.

```
/**
 * The AccessUDDI class implements a simple application
 *  that connects to Microsoft's UBR inquiry node,
 *  searches for service providers that have the string
 *  "abc" in their name and displays to the standard
 *  output the business name, the business description,
 *  and the names of all services provided by that
 *  business.
 */

import org.uddi4j.client.UDDIProxy;
import org.uddi4j.datatype.Name;
import org.uddi4j.response.BusinessInfo;
import org.uddi4j.response.BusinessList;
import org.uddi4j.response.ServiceInfo;
import java.util.Vector;

public class AccessUDDI
{
   public static void main ( String[] args )
   {
      int i = 0;
      int j = 0;

      UDDIProxy proxy = new UDDIProxy ();
      try
      {
         // Set the inquiryURL
         proxy.setInquiryURL
            ( "http://uddi.microsoft.com/inquire" );

         // Look for names that include "abc"
         Vector names = new Vector ();
         names.add ( new Name ( "%abc%" ) );
```

Figure 4-6 Using UDDI4J to access an UDDI Registry to print out 21 providers that include the string "abc" in their names.

```
          // Search the UDDI registry
          BusinessList results =
             proxy.find_business (
                names,
                null,
                null,
                null,
                null,
                null,
                21 );

          Vector businessInfoVect = results.getBusinessInfos
().getBusinessInfoVector ();

          System.out.println ( "Results are:" );

          for ( i = 0 ; i < businessInfoVect.size () ; i++ )
          {
             BusinessInfo businessInfo = ( BusinessInfo )
businessInfoVect.elementAt ( i );

             System.out.println ( "\nName: " +
businessInfo.getNameString () );
             System.out.println ( " ... Description: " +
businessInfo.getDefaultDescriptionString () );

             Vector serviceInfoVect =
businessInfo.getServiceInfos ().getServiceInfoVector ();
             for ( j = 0 ; j < serviceInfoVect.size () ; j++ )
             {
                ServiceInfo servInfo = ( ServiceInfo )
serviceInfoVect.elementAt ( j );
                System.out.println ( " ... Service Name: " +
servInfo.getNameString () );
             }
          }
       }
       catch ( Exception e )
       {
          e.printStackTrace ();
       }
    }
}
```

Figure 4-6 Using UDDI4J to access an UDDI Registry to print out 21 providers that include the string "abc" in their names (continued).

Once the results of the search are returned from the UDDI registry, additional method calls are used to extract the business name, business description, and service names for all matching businesses. This information is then displayed on the standard output. Figure 4-7 shows a selected subset of the output of the application shown in Figure 4-6.

```
Results are:

Name: abc
 ... Description: null

Name: ABC Corporate Services
 ... Description: A travel services company serving the agent
and hotel segments of the industry.
 ... Service Name: Traveler's Emergency Service System (TESS)
 ... Service Name: Premier Hotel Program (PHP)
 ... Service Name: Global Connect

Name: abc Enterprise
 ... Description: test object
 ... Service Name: Deutsche Telekom Productshow
 ... Service Name: Deutsche Telekom Shopping
 ... Service Name: Deutsche Telekom T-Mobil
 ... Service Name: Deutsche Telekom T-Online

Name: abc inc
 ... Description: test desc

Name: ABC Insurance
 ... Description: null

Name: ABC Microsystems
 ... Description: ?~~ ?? ?? ??...???...
 ... Service Name: <New Service Name>

Name: ABC Music
 ... Description: null
 ... Service Name: List Instruments

Name: ABC travel agency
 ... Description: travel buses for goa,bombay,delhi.

Name: abc123
 ... Description: null
 ... Service Name: bogus service

Name: CompanyABC
 ... Description: null

Name: IntesaBci Sistemi e Servizi
 ... Description: IntesaBci Sistemi e Servizi co-ordinate all of
Bank IntesaBci's operations with regard to the development and
management of IT and teleçommunication systems
 ... Service Name: Home Page
```

Figure 4-7 A subset of the result of running the application shown in Figure 4-6.

Looking at the output of Figure 4-7, we can see some of the positive as well as some of the negative points of using the UDDI UBR. First, there are many service providers available within the UBR providing an even larger number of services. These are global providers, and some only offer their services in certain locations. Many of the fields of service providers or services are either unfilled or filled inappropriately. Moreover, many of these service providers or services are either non-existent or simply test deployments.

The UBR is a powerful resource that brings together thousands of providers and services in one easy-to-access location. Sifting through this large (and constantly growing) list to weed out useful providers and services from those that are less than useful (or completely useless) is the difficult part. Although client-side packages such as UDDI4J make developing programs to access and interact with UDDI registries easier, the more important difficulty still remains: how to select the right service and service provider for a given task.

How UDDI Is Playing Out

Now that we have an understanding of the need that UDDI aims to fill, some of the core data structures of UDDI, as well as the variety of the means of communicating with an UDDI registry, it's worth taking a step back to see how UDDI is really playing out. How UDDI will truly be used by companies will determine how, when, where, and why businesses will register their Web services.

Up until now our discussion of UDDI has focused on its analogous behavior with standard telephone directory books: UDDI provides a listing of businesses and the services each business offers as well as a means of searching and discovering Web services to use within consuming applications. Since this usage of UDDI is during the design of applications, it can be referred to as the *design-time* use.

But, will people really use the UDDI APIs during design time? Are people using it today? The answer is not really, and it does not look like it'll change any time in the foreseeable future. Most developers don't programmatically search UDDI for Web services to consume.

Will this change in the future? Most likely not, because selecting which service to consume is difficult. It's not technical issues, but instead business and strategic issues that make the selection process difficult.

In selecting a Web service to use, there may exist business relationships and legal agreements that have to be honored. This may sometimes involve selecting a technically inferior service in order to meet such obligations. There may be pending customer deals that can be closed by using a particular vendor's Web services. A company may attempt to pressure another company by withholding patronage of the latter company's Web services.

Basically business, strategic, and sometimes political issues come into the service selection process. Replacing human intervention through a programmatic API is usually insufficient, and oftentimes grossly so. Because of the wide mix of issues that are often involved, technologists alone will also be insufficient. Accordingly, business analysts, consultants, and other such people (possibly in conjunction with technologists) will usually be responsible for the Web services selection process. These business analysts and consultants will not use the direct programmatic interface of UDDI to search for available services, but instead will use more human-friendly means. These include Web services portals, the home pages provided by some of the UBR node operators, and standard search engines. Of course, word-of-mouth and other such non-technical means will also be prevalent. So, for all intents and purposes, UDDI's programmatic API will probably play a minor role during the design of applications.

If not in the design of applications, where will UDDI play a larger and more prominent role? Although seldom mentioned and even less understood, UDDI has a role larger than just at design time; UDDI is also useful at run time.

UDDI and Lifecycle Management

To understand the usefulness of UDDI at run time, consider the issues that developers and companies have to grapple with after they have developed a Web service or an application that consumes Web services.

Once a Web service has been developed and deployed, it not only has an interface specification but also a network location (usually a URL) associated with it. Over time, the deployment that had sufficed when the service was new and relatively under used, may require changes. This could include migration of the service to a new server. Multiple geographical mirror servers may also be deployed as the need to scale the service increases, or a new server location may be launched while the original one is taken offline for maintenance. The organization or division maintaining the Web service may be relocated or sold, thereby necessitating an update to its access endpoint information. How can these changes be propagated to applications that have already been designed to consume the original Web service? Without appropriate dissemination of such changes, applications consuming the original service can malfunction or produce erroneous transactions.

An application that consumes Web services has to contend with similar issues. Once an application has been written to use a specific Web service for a particular part of its functionality, the application's capability with respect to that part of its functionality is dependent on the Web service. If the Web service goes down or is unavailable for some time, that part of the application will also not function, possibly causing erroneous behavior throughout the application.

Applications based on Web services need a mechanism to stay updated with the latest access endpoint information, including changes to older endpoints, for a particular Web service. It is precisely in this need for lifecycle management of applications and Web services where UDDI can play a critical role. Web services need to disseminate changes to applications that call them. Applications need to be made aware of these changes. UDDI can play the runtime broker or middleman in handling and propagating these changes. The steps in this lifecycle management scenario proceed as follows:

1. Locate a Web service that fulfills the application's needs using whatever means that are useful, including portals, service aggregators, or programmatically with an UDDI registry directly.
2. If the Web service was not initially discovered within an UDDI registry, locate the same service within an UDDI registry and save (e.g., in a database) the `bindingTemplate` information.
3. Develop the application to consume the Web service using the information from the saved `bindingTemplate` information.
4. If the Web service call fails or exceeds an application-specified time-out, query the UDDI registry for the latest information on that Web service.
5. In case the original Web service call failed, compare the latest binding information for that Web service with the saved information. If the latest binding information for the Web service is different from the saved information, then save the new binding information, and retry the Web service call.
6. In the case that the original Web service call exceeded a time-out, compare the latest binding information for that Web service with the saved information. If the information is different or newer access endpoints are available, select another endpoint. The selection procedure may be manual in which the application allows the user to manually choose, or it may be automatic.

In this scenario, UDDI plays a critical role in maintaining the reliability and quality-of-service of both applications and the Web services they consume throughout their lifecycle.

The subset of a simple application that demonstrates the use of UDDI at runtime is shown in Figure 4-8. This code snippet uses the UDDI4J client-side Java API, does not do any error checking and also assumes a simple binding described by the UDDI registry.

Once a Web service invocation fails, the application tries to determine whether the binding information for the service has changed. If it has changed, the new binding information can be incorporated into the Web service call and the service can be retried. Otherwise, an error has to be thrown notifying the user that the service is unavailable.

The application begins by retrieving the binding information for the saved binding key by using the `get_bindingDetail` method. From the `BindingDetail` object, the program extracts the latest access point URL for the Web service. By comparing the latest access point information stored in `newAccessPoint` with the original access point information stored in `accessPoint`, the program is able to determine whether the cause of the service invocation failure was due to a change in the service's binding information. If new binding information is available, the program updates the `accessPoint` variable with the latest information and sets the `retryService` to true indicating that the service call can now be retried. If no new binding information is available, then the service is unavailable and there is no need to retry the service call. The program sets `retryService` to false.

```
// The Web service invocation failed, so check to see
//  whether new binding information is available. If so,
//  retry the Web service call.

BindingDetail bd = proxy.get_bindingDetail ( bindingKey );

Vector btvect = bindingDetail.getBindingTemplateVector ();
BindingTemplate bt = ( BindingTemplate )
                                 btvect.elementAt ( 0 );
newEndpoint = bt.getAccessPoint ().getText ();

if ( thisEndpoint.equalsIgnoreCase ( newEndpoint ) )
{
   // In this case, the endpoint information has changed
   //  so we should retry the Web service invocation with
   //  with the new endpoint

   thisEndpoint = newEndpoint;
   retry = true;
}
else
{
   // In this case, the endpoint information has not
   //  changed so there no reason to retry the Web
   //  service invocation

   retry = false;
}
```

Figure 4-8 Retrying Web service invocations based on dynamic UDDI information.

UDDI and Dynamic Access Point Management

As we've already alluded to, UDDI at runtime can be used not only to get an updated access point URL, but also to dynamically manage and select the most appropriate access point. Oftentimes, a Web service will be deployed on multiple machines that have different characteristics. These characteristics can differ by geographical locations and amount of server resources, including type of network connectivity.

Usually, this variety of service deployments is dynamic, that is, the Web service is initially deployed on a single server. Later, as the service becomes more popular and demand increases, additional access points are deployed. These deployments may be a cluster of servers in close proximity to each other, a geographically distributed set of servers, or both.

A client application that consumes the Web service may have been developed before the additional access points were deployed. Or the best service at the time the application was developed is no longer the best or the most appropriate. For example, the client application may have been developed in one country and later used in another country. Hardwiring the service access point to the one that was selected at design time (in a country other than where the application is being used) will needlessly increase the latency of the service invocation. Mobile applications are most vulnerable to this situation as the application may be best suited to a different access point as the mobile user moves from location to location.

Managing the Web service access points used by a client application becomes increasingly important. It's not that the application will not work with a hardwired access point (assuming the access point remains operational for the life of the application). Instead, the application may potentially work better.

Selecting and managing access points is analogous to downloading files from different mirror sites. A user can certainly download all her content from a single site. But, by judicious selection of different mirror sites, the user can achieve improved performance. The selection of Web service access points can be manual in which the application user is given the ability to choose the actual access point, or the application may automatically select an access point by consulting an UDDI registry. Alternately, the user may specify the high-level characteristics and metrics that are most important to him, with the application using those characteristics as hints in determining the most appropriate service access points. Refer to Chapter 10 for a more in-depth discussion of quality-of-service issues and Web services.

The careful reader will have recognized that some of the benefits of UDDI at run time can also be obtained from alternate means. Using databases, configuration files and other registries are some obvious alternatives. Although other solutions are possible, using UDDI is preferable as it is a standards-based solution with tremendous support from the software industry. The most important benefit of using a standards-based solution with industry-wide support such as UDDI is that almost all Web services can be used. With non-standard solutions, the Web service vendor must also publish its information using the same means used by the application vendor. In cases where a single vendor owns and has administrative control over both the services and the applications, such a solution is manageable. When the service vendors and application vendors are different, a standard solution fosters the use of a variety of Web services.

Figure 4-9 summarizes the use of UDDI at both the design time as well as the run time of Web services-based applications. As the figure depicts, interactions with UDDI at design time will usually include manual intervention from a variety of sources, such as business analysts, consultants, strategists, and technologists to determine the most appropriate Web service. It is important to note that the "most appropriate" service may not be the highest performance service. At run time, however, there is plenty of opportunity to leverage the direct programmatic access of UDDI to build applications that dynamically select the "best" service deployment (from the "most appropriate" service that was determined in the design phase).

In this section, we have discussed just a few uses of UDDI at application run time. Many more uses are possible. In particular, as UDDI matures and more information is made available through UDDI registries, additional opportunities to build more robust and flexible applications will emerge. When developing applications that consume Web services, if developers find themselves hardwiring information particular to a specific service into their applications, alarm bells

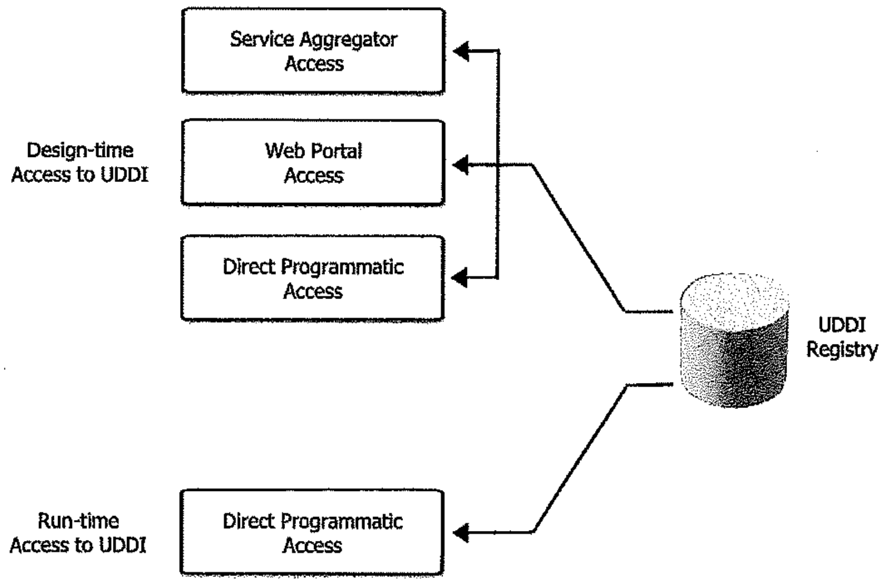

Service Aggregator Access
Design-time Access to UDDI
Web Portal Access
Direct Programmatic Access
UDDI Registry
Run-time Access to UDDI
Direct Programmatic Access

Figure 4-9 The use of UDDI at both design time and run time.

should immediately sound. They must ask themselves whether it is possible to eliminate the direct dependency on a particular service with an indirect and flexible "brokered" access through UDDI.

Summary

UDDI is an industry standard for a platform-independent and flexible means of describing, discovering and integrating services as well as the businesses that provide the services. As we have seen, UDDI has many similarities to telephone books, and provides users a means to search for Web services as well as service provider businesses.

The UBR is a global implementation of the UDDI specification and provides a publicly accessible registry of Web services. Currently, IBM, Microsoft, SAP, and NTT provide UBR nodes where users can register their Web services and make them available to a global market.

Although the UDDI specification provides a programmatic API to publish Web services to a UDDI registry and also to inquire about which services and service providers are available, most service selection issues at design time will require human intervention, thus reducing the usefulness of an automatic, programmatic interface. The business, strategic, and sometimes political issues that come into the service selection process will usually require business analysts and strategic consultants to play a critical role in the service selection process. Accordingly, during the design of an application, more human-friendly means to service selection including aggregation portals such as XMethods, Internet search engines such as Google, word of mouth, and UBR home pages, will be critical.

Instead of its much-hyped role at application design time, UDDI plays a more useful role at application runtime. Applications based on Web services need a mechanism to stay updated with the latest access endpoint information for a particular Web service. Conversely, Web services need a means to broadcast to applications that are already consuming them additional capabilities and resources. UDDI registries and the global UBR implementation provide such capabilities, and can play a critical role in the lifecycle of Web services and the applications that consume them.

UDDI is an important technology with useful capabilities. These capabilities must be properly positioned within the limitations of businesses and the usual operations of partner interactions. As we have discussed, with the right positioning, UDDI forms a core piece of the enterprise Web services platform.

Architect's Notes

- Today, most Web services are discovered through non-programmatic means using manual, human intervention. Similar to the way companies scrutinize potential partner companies prior to committing to a strategic relationship, selecting services to use within enterprise applications requires significant due diligence. Manual intervention by business analysts, consultants, and others familiar with the company's business and strategic needs will almost always be required for selecting services. The most common sources of locating Web services are existing business partners, UBR home pages, service aggregators such as XMethods, or standard Web search engines such as Google.
- The UDDI Business Registry (UBR) is a distributed, public registry containing thousands of service providers and even more services. Sifting through this large (and constantly growing) list to weed out useful providers and services from those that are less than useful (or completely useless) is the biggest drawback and the major difficulty of using such public resources.
- Enterprise UDDIs and other such private (or semi-private) UDDIs that support and facilitate easy access to Web services and other resources within an organization will provide direct value. Typical use cases for UDDI within organizations will be to support and manage reuse of programmatic resources throughout an extended enterprise, as well as to dynamically configure and customize an application by changing attributes within the UDDI.
- Using client-side API packages such as UDDI4J and Microsoft's UDDI SDK facilitate developing programs that access UDDI registries, and also insulate applications from specification changes and differences between various registries.

- UDDI plays a critical and potentially larger role during the run time of applications. Typically, UDDI is seen as a means of discovering services at design time. UDDI also provides a convenient means to manage the lifecycle of Web services as well as the applications that consume them. Changes to information about a Web service can be pushed onto an UDDI registry, and applications that consume that service can be developed to be more reliable and robust by simply querying the UDDI registry for changes upon any invocation failures or other unexpected behavior.