The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Service Design Patterns

Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services

Robert Daigneau

With a Contribution by
Ian Robinson

Forewords by
Martin Fowler and Ian Robinson

Web Service API Styles

RPC API (18)	How can clients execute remote procedures over HTTP?
Message API (27)	How can clients send commands, notifications, or other information to remote systems over HTTP while avoiding direct coupling to remote procedures?
Resource API (38)	How can a client manipulate data managed by a remote system, avoid direct coupling to remote procedures, and minimize the need for domain-specific APIs?

Client-Service Interaction Styles

Request/Response (54)	What's the simplest way for a web service to process a request and provide a result?
Request/Acknowledge (59)	How can a web service safeguard systems from spikes in request load and ensure that requests are processed even when the underlying systems are unavailable?
Media Type Negotiation (70)	How can a web service provide multiple representations of the same logical resource while minimizing the number of distinct URIs for that resource?
Linked Service (77)	Once a service has processed a request, how can a client discover the related services that may be called, and also be insulated from changing service locations and URI patterns?

Request and Response Management

Service Controller (85)	How can the correct web service be executed without having to write complex parsing and routing logic?
Data Transfer Object (94)	How can one simplify manipulation of request and response data, enable domain layer entities, requests, and responses to vary independently, and insulate services from wire-level message formats?
Request Mapper (109)	How can a service process data from requests that are structurally different yet semantically equivalent?
Response Mapper (122)	How can the logic required to construct a response be reused by multiple services?

Web Service Implementation Styles

Transaction Script (134)	How can developers quickly implement web service logic?
Datasource Adapter (137)	How can a web service provide access to internal resources like database tables, stored procedures, domain objects, or files with a minimum amount of custom code?
Operation Script (144)	How can web services reuse common domain logic without duplicating code?
Command Invoker (149)	How can web services with different APIs reuse common domain logic while enabling both synchronous and asynchronous request processing?
Workflow Connector (156)	How can web services be used to support complex and long-running business processes?

Web Service Infrastructures

Service Connector (168)	How can clients avoid duplicating the code required to use a specific service and also be insulated from the intricacies of communication logic?
Service Descriptor (175)	How can development tools acquire the information necessary to use a web service, and how can the code for Service Connectors be generated?
Asynchronous Response Handler (184)	How can a client avoid blocking when sending a request?
Service Interceptor (195)	How can common behaviors like authentication, caching, logging, exception handling, and validation be executed without having to modify the client or service code?
Idempotent Retry (206)	How can a client ensure that requests are delivered to a web service despite temporary network or server failures?

Web Service Evolution

Single-Message Argument (234)	How can a web service with an *RPC API* (##) become less brittle and easily accommodate new parameters over time without breaking clients?
Dataset Amendment (237)	How can a service augment the information it sends or receives while minimizing the probability of breaking changes?
Tolerant Reader (243)	How can clients or services function properly when some of the content in the messages or media types they receive is unknown or when the data structures vary?
Consumer-Driven Contracts (250)	How can a web service API reflect its clients' needs while enabling evolution and avoiding breaking clients?

Service
Design Patterns

Service Design Patterns

Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services

Robert Daigneau

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data
Daigneau, Robert.
Service design patterns : fundamental design solutions for SOAP/WSDL and restful Web services / Robert Daigneau.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-321-54420-9 (hardcover : alk. paper)
ISBN-10: 0-321-54420-X (hardcover : alk. paper)
1. Web services. 2. Web site development. 3. Simple Object Access Protocol (Computer network protocol) I. Title.
TK5105.88813.D35 2012
006.7'8—dc23
2011033436

ISBN-13: 978-0-321-54420-9
ISBN-10: 0-321-54420-X

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, October 2011

For John, Alice, Heather, and Michelle

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

by Martin Fowler

One of the inevitable truisms of enterprise applications is that they are not islands. You may be focused on solving a particular business problem, but in order to do that you won't be able to capture all the data you need yourself, or develop all the processing. Even if you had the time, that data and processing is being done elsewhere, and duplication is both wasteful and leads to messy inconsistencies. As a result, almost all enterprise applications need to communicate with other applications. These foreign systems are often not in the same organization, but are provided by some third-party organization.

For many years, one of the hardest parts of this kind of collaboration was just to get some kind of communication path. Often these applications were written on different platforms, with different languages, on different operating systems supporting different communication protocols. But in the past decade, the web has appeared as a solution to the connection problem. Almost all systems can open port 80 and talk text over it.

But that still leaves many questions around how they should talk. Should they use an RPC-style API, a message-oriented API, or this fashionable REST stuff? Should logic be embedded in services directly or delegated to underlying objects? How can we change services that are already in use without breaking clients?

Generally in my series, the books have featured topics that haven't been covered much elsewhere, but there have already been too many books about various aspects of web services. As a result, when a draft of Robert's book came to me across the ether, I didn't think I would be interested in it. What changed my mind was that it brings together these key questions into a single handbook, in a style that I like to see in a technical book that's worth the effort of reading.

First, he takes the approach of breaking up the topic area into patterns, so we have vocabulary to talk about these topics. Then he goes into each pattern, explaining how each one works and how to choose between them. As a result, you are able to see the various approaches to web service design and decide what will work for you in your context. He provides code examples, so you can see how these patterns might work in practice, yet the patterns are general enough to apply to many technology stacks.

The result is a book that collects the important design decision points for using web services in a style that focuses on principles that are likely to be valuable despite changes in technology.

Martin Fowler
http://martinfowler.com

Foreword

by Ian Robinson

Distributed application development often starts well. And just as often it ends badly. *Point*, *add web reference*, *click:* That's the sound of a developer pointing a loaded client at your carefully crafted service interface. By substituting tooling for design, we somehow turned all that loose coupling into plain irresponsible promiscuity; come release time, we all have to join the lockstep jump.

In a more cautious era, we'd have just said: "No. Don't distribute." And in large part that advice still holds true today. A layer is not a tier. Blowing a three-layered application architecture out to distributed proportions is foolishness writ large, no matter how many open standards you implement.

But today's applications are rarely islands. Where a business's capabilities are scattered across organizational boundaries, so too are the systems that automate them. Some form of service orientation, both within and between companies, is necessary if we are to support the distributed nature of the modern supply chain.

The web, or rather, the technology that underpins the web, has proven enormously resourceful in this respect. Whether or not you're aware of—or even carelessly indifferent to—the web's prominent place in the history of distributed systems, there's inevitably something of the web about a sound majority of the services you've built or used. For all its purported transport agnosticism, SOAP, in practice, has tended to ride the HTTP train. Hidden, but not forgotten, the web has shouldered the services burden for several years now.

When we look at the web services landscape today, we see that there are at least three ways to accommodate the web in the software we build. The web has succeeded not because of the overwhelming correctness of its constituency, but because of its tolerance for the many architectural styles that inhabit and sometimes overrun its borders. Some services and applications are simply *behind* the web. They treat the web as an unwelcome but nonetheless necessary narrow gateway through which to access objects and procedures. Adjust your gaze, and you'll see that some services are *on* the web; that is, they treat HTTP not as a brute transport, but rather as the robust coordination and transfer protocol described in RFC 2616. Last, you'll see some (very few) that are *of* the web. These use the web's founding technologies—in particular, URIs and HTTP and generalized hypermedia representation formats such as HTML—to present a web of data, including data that describes how to access and manipulate more data, to consumers.

This book brings together the need for caution and defensive design when distributing systems with the several ways of using the web to enable distribution. As a compendium of sound strategies and techniques, it rivals the hard-won experience of many of my friends and colleagues at ThoughtWorks. It's a book about getting things done on the web; it's also a book about not backing yourself into a corner. By balancing the (necessary) complexity of shielding a service's domain and data from that army of cocked clients with the simplicity that begets internal quality and service longevity, it may just help you avoid the midnight lockstep deployment.

Ian Robinson

Preface

When I started working on this book I wasn't entirely sure what SOA and REST were. I knew that I wasn't the only one who felt this way. Most discussions on these topics were rife with ambiguity, hyperbole, misinformation, and arguments that appealed to emotion rather than reason. Still, as a developer who had struggled with distributed object technologies, I was fascinated by web services. I saw them as a pragmatic way to integrate systems and reuse common business logic.

Since then, REST has gained significant momentum, WS* services have established a solid foothold, and SOA was proclaimed dead [Manes]. Through it all, my fascination with web services never waned. As mobile, cloud, and Software-as-a-Service (SaaS) platforms cause software to become increasingly distributed, the importance of web services will only continue to increase. We live in exciting times indeed!

What Is This Book About?

This book is a catalogue of design solutions for web services that leverage SOAP/WSDL or follow the REST architectural style. The goal was to produce a concise reference that codifies fundamental web service design concepts. Each pattern describes a known and proven solution to a recurring design problem. However, the patterns are not meant to be recipes that are followed precisely. In fact, a given pattern might never be implemented in exactly the same way twice. This catalogue also doesn't invent new solutions. Rather, the patterns in this book were identified over long periods of time by developers who noticed that certain problems could be solved by using similar design approaches. This book captures and formalizes those ideas.

Services can be implemented with many different technologies. SOA practitioners, for example, often say that technologies as diverse as CORBA and

DCOM, to the newer software frameworks developed for REST and SOAP/WSDL, can all be used to create services. This book focuses exclusively on web services. Unfortunately, this term is somewhat overloaded as well. Some use it to refer to any callable function that uses WSDL. The term has also been used to describe RESTful services (re: [Richardson, Ruby]). This book uses the term *web service* to refer to software functions that can be invoked by leveraging HTTP as a simple transport over which data is carried (e.g., SOAP/WSDL services) or by using HTTP as a complete application protocol that defines the semantics for service behavior (e.g., RESTful services).

Who Is This Book For?

This book is aimed at professional enterprise architects, solution architects, and developers who are currently using web services or are thinking about using them. These professionals fall into two distinct groups. The first group creates software products (e.g., commercial, open source SaaS applications). The second develops enterprise applications for corporate IT departments. While this catalogue is tailored for software professionals, it can also be used in academia.

What Background Do You Need?

Pattern authors often provide code examples to illustrate design solutions. Most catalogues aren't meant to be platform-specific, but the author must still choose which languages, frameworks, and platforms to use in the examples. While a plethora of new languages have become popular in recent years, I decided to use Java and C# for two reasons. First, these languages have a significant market share (i.e., a large installed base of applications) and are quite mature. Second, most readers probably use or have used these languages and are therefore familiar with their syntax. I will assume that the reader has an intermediate to advanced understanding of these languages and of several object-oriented programming (OOP) concepts.

The patterns in this catalogue make heavy use of a few web service frameworks popular with Java and C# developers. These frameworks encapsulate the most common functions used by web service developers. This book does not identify the patterns used within these frameworks. Instead, it identifies patterns that developers use when leveraging these frameworks to build web services. Here are the frameworks that are used in this book:

- SOAP/WSDL frameworks:
  - The Java API for XML Web Services (JAX-WS)
  - Apache CXF
  - Microsoft's Windows Communication Foundation (WCF)
- REST frameworks:
  - The Java API for RESTful Web Services (JAX-RS)
  - Microsoft's WCF
- Data-binding frameworks:
  - The Java Architecture for XML Binding (JAXB)
  - Microsoft's `DataContractSerializer` and other serializers (e.g., `XmlSerializer`)

It is assumed that the reader will at least have a basic acquaintance with the following:

- JavaScript Object Notation (JSON)
- Extensible Markup Language (XML)
- XML Schema Definition Language
- XML Path Language (XPath)
- Extensible Stylesheet Language Transformation (XSLT)
- The Web Services Description Language (WSDL)

Organization of This Book

Following a general introduction in Chapter 1, the patterns in this catalogue are grouped into six chapters.

- **Chapter 2, Web Service API Styles:** This chapter explores the primary API styles used by web services. The ramifications of selecting the right style cannot be underestimated because, once a style is chosen, it becomes very hard to change direction.

- **Chapter 3, Client-Service Interactions:** This chapter presents the foundations for all client-service interactions. These patterns may be used with any service design style. Given an understanding of these patterns, you can devise complex conversations in which multiple parties exchange data about a particular topic over short or extended periods of time.
- **Chapter 4, Request and Response Management:** Software applications are frequently organized into layers that contain logically related entities. This chapter identifies the common *Service Layer* [POEAA] entities that are used to manage web requests and responses. The intent of these patterns is to decouple clients from the underlying systems used by the service.
- **Chapter 5, Web Service Implementation Styles:** Services may be implemented in various ways. They may have intimate knowledge of resources such as database tables, they may coordinate the activities of an Object Relational Mapper (ORM) or direct calls to legacy APIs, or they may forward work to external entities. This chapter looks at the ramifications of each approach.
- **Chapter 6, Web Service Infrastructures:** Certain tasks are so generic that they can be used over and over again. This chapter discusses some of the most common and basic infrastructure concerns pertinent to client and service developers. A few patterns common to corporate SOA infrastructures are also reviewed.
- **Chapter 7, Web Service Evolution:** Developers strive to create services that will remain compatible with clients which evolve at different rates. This goal, however, is quite difficult to achieve. This chapter reviews the factors that cause clients to break and discusses two common versioning strategies. You'll also see how services can be augmented to meet client requirements while avoiding a major software release.

Supporting information is provided in the Appendix, Bibliography, and Glossary.

The Pattern Form Used in This Book

There are many ways to present patterns, from the classic style of Christopher Alexander [Alexander] to the highly structured forms of the Gang of Four [GoF] and *Pattern-Oriented Software Architecture* [POSA] books. The convention used in this book was influenced by the Alexandrian form and the style used in *Enterprise Integration Patterns* [EIP]. Hopefully you will find that the conversational style makes the patterns easy to read. Only a few recurring headers are used to demarcate content. Each design pattern is described using the following conventions.

- **Pattern name:** The pattern name describes the solution in a few words. The name provides a handle or identifier for the solution, and is a part of the larger pattern language presented in the book. The goal was to use evocative names that can be easily understood and used in everyday conversations. Many of the pattern names in this book are already quite common.
- **Context:** The context follows the pattern name and is expressed in no more than a few sentences. It identifies the general scenario in which the pattern might apply. Of course, all of these patterns apply to web services, but some are only relevant to certain situations. This section may refer to other patterns to help set the context.
- **Problem:** The problem to solve is stated as a single question. You should be able to read the problem and quickly determine if the pattern is relevant to the design challenge you are facing. This section is marked off between two horizontal bars.
- **Forces:** The forces provide more detail on the problem. This section, which follows the problem definition, explores some of the reasons why the problem is difficult to solve and presents alternative solutions that have been tried but may not work out so well. The goal of this narrative is to naturally lead you to the solution.
- **Solution summary:** This section provides a brief description of the design solution, in a few sentences. Despite its terseness, you should be able to quickly understand how the problem can be solved. The solution summary typically describes the primary entities that comprise the design, their responsibilities and relationships, and the way they work together to solve the problem. The solution is not meant to be an absolute prescription that has one and only one implementation. Rather, it should be viewed as a general template that can be implemented in many different ways. The written summary is usually accompanied by a diagram to supplement the narrative. The primary mechanisms used in this book include sequence diagrams and class diagrams. In some cases, the solution is modeled through nonstandard graphical depictions. This section is demarcated by

two horizontal bars, just like the problem section. The intent was to make it easy for readers to quickly find the problem and solution summary.

- **Solution detail:** This section presents several aspects of the solution in a prosaic style. It expands on the solution summary to explore how the primary elements of the solution are employed in order to solve the problem and resolve the forces. Since every solution has benefits and drawbacks, this section reviews the consequences and additional factors you may need to consider. I also point out related patterns that may be found elsewhere in this book or in other pattern catalogues. This is done for a variety of reasons. Some of the patterns in this book are actually specializations of existing patterns, and I felt that it was only right to acknowledge the original source. Other patterns are complementary to the pattern being discussed, or may be considered as an alternative.
- **Considerations:** This section discusses additional factors you may need to consider when using the pattern. Such factors include design considerations, related technologies, and a variety of other pertinent topics. Bulleted lists are used to help the reader skim this section and hone in on specific topics of interest. This section does not occur in each and every pattern.
- **Examples:** This section is meant to supplement the prior sections. You should be able to understand the essence of a pattern without having to read this section. Some patterns offer several examples to help you understand the many ways in which the pattern may be implemented. Other patterns only provide a few examples to facilitate understanding.

  The examples in this book take many forms. The most common form is Java and C# code. Other examples use XML, JSON, XSD, and WSDL. All attempts were made to keep the examples as simple as possible, so a number of things were left out (e.g., exception-handling blocks, thread management, most queuing and database-related logic, etc.). In some cases, the examples only include enough code to convey the basic idea of the pattern. In other cases, the code examples provide much more detail because I felt that omitting such detail would have left the reader with too many questions.

  Please note that this is not meant to be a book on how to use a specific API; there are many great books out there for such matters. Rather, these code samples are provided to deepen your understanding of abstract design solutions. Furthermore, just as the pattern descriptions provide a template, so too do the code examples. This means that you probably won't want to copy any code verbatim.

Pattern Categories Not Covered

A vast array of topics has been included under the umbrella of service design. However, many of these subjects are quite deep and have already been covered extensively in other works. The content of this catalogue has therefore been constrained to only include the most fundamental solutions relevant to web service design. The following topics have been avoided or only covered lightly.

- **Enterprise integration patterns:** While web services provide a great way to integrate disparate applications, the topic of integration is exceedingly deep. Hohpe and Woolf's book, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* [EIP], does a great job of showing how integration can occur through middleware solutions that primarily leverage queuing technologies. This book builds on and refers to many of their patterns.
- **Workflow/orchestration:** Workflow technologies provide the means to define the flow of execution through a set of related activities managed by a central controller. Workflows are frequently triggered by web services and often interact with external web services to send or receive data. Workflows may be relatively short in duration (i.e., a few seconds), or may transpire over days, weeks, or even months. The subject of workflow is far beyond the scope of this book, and is addressed by such catalogues as http://workflowpatterns.com [van der Aalst, et al.].
- **Event-driven architecture:** An alternative to the "command and control" architectural style exemplified by workflows is event-driven architecture (EDA). With EDA, there is no centralized controller. Instead, clients and services communicate with each other in a much more fluid, dynamic, and often unpredictable way when specific events occur within their respective domains. Sometimes the rules for EDA are described through choreography. For more information on this topic I recommend the following:

  www.complexevents.com/category/applications/eda/
- **Choreography:** Choreography, like EDA, is not a "command and control" architectural style. Instead, it suggests that parties adopt a rules-based approach that declares the sequence of allowed exchanges between parties as if seen by an external observer. This concept has yet to see wide adoption.

- **Security:** Matters such as authentication, authorization, data confidentiality, data integrity, nonrepudiation, techniques used to harden network infrastructures, and other security concerns are not discussed in any detail as these subjects are incredibly deep and have been covered extensively in other works.

Supporting Web Site and contact information

Companion information for this book may be found at

www.ServiceDesignPatterns.com

Acknowledgments

I'd like to acknowledge everyone who has assisted in this project. I could not have done this without you.

I am indebted to a number of people from the Pattern Languages of Programming Conference (PLoP) held in 2008. Kyle Brown played an instrumental role by pointing me toward an effective pattern form. He instilled in me a strategy for setting up the problem, forces, and constraints. I'd also like to thank my PLoP 2008 workshop mates for their feedback on early versions of a few patterns and my writing style. This group included Paul Austrem, Philipp Bachmann, Geert Monsieur, Mark Mahoney, Lise Hvatum, Bobby Woolf, and John Liebenau.

Bobby Woolf, Rebecca Wirfs-Brock, and Michael Stiefel volunteered to review early versions of a few chapters. Your feedback helped steer me in the right direction. John Liebenau and Uwe Zdun also offered help in those early days.

My biggest thanks go to my official team of technical reviewers. Jim Webber was incredibly gracious with his time and proved to be a reliable REST resource. Eric Newcomer offered thoughts on how to fine-tune the book's scope. Marc Hadley kept me honest on several Java Specification Requests (JSRs), and provided great input in several other areas. Scott Ambler offered several suggestions to help make the patterns more readable. Elliotte Rusty Harold's expertise is world renowned, and his critiques were thought provoking. Ian Robinson made a huge contribution by offering a very important pattern to the book. Rebecca Wirfs-Brock, more than anyone else, recognized my strengths and weaknesses. Her measured feedback was invaluable. I am, of course, extremely grateful to Martin Fowler for accepting me into this prestigious series and working with me to sharpen the book's content.

Last but not least, I must thank my editorial team, Chris Guzikowski, Raina Chrobak, Chris Zahn, Julie Nahil, and Audrey Doyle. Without your patience, support, and a little prodding, I would have never reached the finish line.

About the Author

Robert Daigneau has more than twenty years of experience designing and implementing applications and products for a broad array of industries from financial services, to manufacturing, to retail and travel. Rob has served in such prominent positions as Director of Architecture for Monster.com, and Manager of Applications Development at Fidelity Investments. He has been known to speak at a conference or two.

Rob can be reached at rob@ServiceDesignPatterns.com.

Chapter 1

From Objects to Web Services

Web services have been put into practical use for many years. In this time, developers and architects have encountered a number of recurring design challenges related to their usage. We have also learned that certain design approaches work better than others to solve certain problems. This book is for software developers and architects who are currently using web services or are thinking about using them. The goal is to acquaint you with some of the most common and fundamental web service design solutions and to help you determine when to use them. All of the concepts discussed here are derived from real-life lessons. Proven design solutions will also be demonstrated through code examples.

Service developers are confronted with a long list of questions.

- How do you create a service API, what are the common API styles, and when should a particular style be used?
- How can clients and services communicate, and what are the foundations for creating complex conversations in which multiple parties exchange data over extended periods of time?
- What are the options for implementing service logic, and when should a particular approach be used?
- How can clients become less coupled to the underlying systems used by a service?
- How can information about a service be discovered?
- How can generic functions like authentication, validation, caching, and logging be supported on the client or service?

- What changes to a service cause clients to break?
- What are the common ways to version a service?
- How can services be designed to support the continuing evolution of business logic without forcing clients to constantly upgrade?

These are just a few of the questions that must be answered. This book will help you find solutions that are appropriate for your situation.

In this chapter, you'll learn what services are and how web services address the shortcomings of their predecessors.

What Are Web Services?

From a technical perspective, the term **service** has been used to refer to any software function that carries out a business task, provides access to files (e.g., text, documents, images, video, audio, etc.), or performs generic functions like authentication or logging. To these ends, a service may use automated workflow engines, objects belonging to a *Domain Model* [POEAA], commercial software packages, APIs of legacy applications, Message-Oriented Middleware (MOM), and, of course, databases. There are many ways to implement services. In fact, technologies as diverse as CORBA and DCOM, to the newer software frameworks developed for REST and SOAP/WSDL, can all be used to create services.

This book primarily focuses on how services can be used to share logical functions across different applications and to enable software that runs on disparate computing platforms to collaborate. A platform may be any combination of hardware, operating system (e.g., Linux, Windows, z/OS, Android, iOS), software framework (e.g., Java, .NET, Rails), and programming language. All of the services discussed in this book are assumed to execute outside of the client's process. The service's process may be located on the same machine as the client, but is usually found on another machine. While technologies like CORBA and DCOM can be used to create services, the focus of this book is on web services. **Web services** provide the means to integrate disparate systems and expose reusable business functions over HTTP. They either leverage HTTP as a simple transport over which data is carried (e.g., SOAP/WSDL services) or use it as a complete application protocol that defines the semantics for service behavior (e.g., RESTful services).

Terminology

Web service developers often use different terms to refer to equivalent roles. Unfortunately, this has caused a lot of confusion. The following table is therefore provided for clarification and as a reference. The first column lists names used to denote software processes that send requests or trigger events. The second column contains terms for software functions that respond or react to these requests and events. The terms appearing under each column are therefore synonymous.

Client	Service
Requestor	Provider
Service consumer	Service provider

This book uses the terms "*client*" and "*service*" because they are common to both SOAP/WSDL services and RESTful services.

Web services were conceived in large part to address the shortcomings of distributed-object technologies. It is therefore helpful to review some history in order to appreciate the motivation for using web services.

From Local Objects to Distributed Objects

Objects are a paradigm that is used in most modern programming languages to encapsulate behavior (e.g., business logic) and data. Objects are usually "fine-grained," meaning that they have many small properties (e.g., `FirstName`, `LastName`) or methods (e.g., `getAddress`, `setAddress`). Since developers who use objects often have access to the internals of the object's implementation, the form of reuse they offer is frequently referred to as white-box reuse. Clients use objects by first instantiating them and then calling their properties and methods in order to accomplish some task. Once objects have been instantiated, they usually maintain state between client calls. Unfortunately, it wasn't always easy to use these classes across different programming languages and platforms. Component technologies were developed, in part, to address this problem.

Components were devised as a means to facilitate software reuse across disparate programming languages (see Figure 1.1). The goal was to provide a means whereby software units could be assembled into complex applications much like electronic components are assembled to create circuit boards. Since developers who use components cannot see or modify the internals of a component, the form of reuse they offer is called black-box reuse. Components group related objects into deployable binary software units that can be plugged into applications. An entire industry for the Windows platform arose from this concept in the 1990s as software vendors created ActiveX controls that could be easily integrated into desktop and web-based applications. The stipulation was that applications could not access the objects within components directly. Instead, the applications were given binary interfaces that described the objects' methods, properties, and events. These binary interfaces were often created with platform-specific interface definition languages (IDLs) like the Microsoft Interface Definition Language (MIDL), and clients that wished to use components frequently had to run on the same computing platform.

Objects were eventually deployed to remote servers in an effort to share and reuse the logic they encapsulated (see Figure 1.2). This meant that the memory that was allocated for clients and distributed objects not only existed in separate address spaces but also occurred on different machines. Like components, distributed objects supported black-box reuse. Clients that wished to use distributed objects could leverage a number of remoting technologies like CORBA, DCOM, Java Remote Method Invocation (RMI), and .NET Remot-

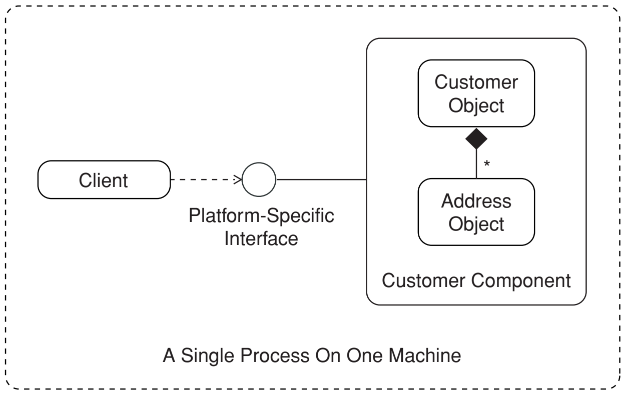

Figure 1.1 *Components were devised as a means to facilitate reuse across disparate programming languages. Unfortunately, they were often created for specific computing platforms.*

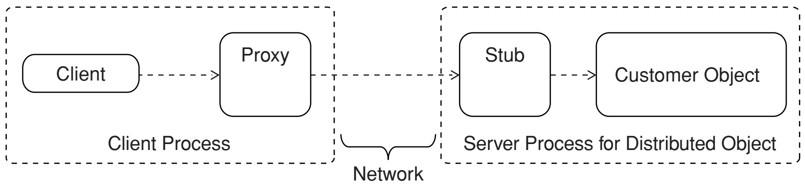

Figure 1.2 *Objects were frequently used in distributed scenarios. When a client invoked a method on the proxy's interface, the proxy would dispatch the call over the network to a remote stub, and the corresponding method on the distributed object would be invoked. As long as the client and distributed object used the same technologies, everything worked pretty well.*

ing. The compilation process for these technologies produced a binary library that included a *Remote Proxy* [GoF]. This contained the logic required to communicate with the remote object. As long as the client and distributed object used the same technologies, everything worked pretty well. However, these technologies had some drawbacks. They were rather complex for developers to implement, and the process used to serialize and deserialize objects was not standardized across vendor implementations. This meant that clients and objects created with different vendor toolkits often had problems talking to each other. Additionally, distributed objects often communicated over TCP ports that were not standardized across vendor implementations. More often than not, the selected ports were blocked by firewalls. To remedy the situation, IT administrators would configure the firewalls to permit traffic over the required ports. In some cases, a large number of ports had to be opened. Since hackers would have more network paths to exploit, network security was often compromised. If traffic was already permitted through the port, then it was often already provisioned for another purpose.

Distributed objects typically maintained state between client calls. This led to a number of problems that hindered scalability.

- Server memory utilization degraded with increased client load.
- Effective load-balancing techniques were more difficult to implement and manage because session state was often reserved for the client. The result was that subsequent requests were, by default, directed back to the server where the client's session had been established. This meant that the load for client requests would not be evenly distributed unless a sophisticated

infrastructure (e.g., shared memory cache) was used to access the client's session from any server.

- The server had to implement a strategy to release the memory allocated for a specific client instance. In most cases, the server relied on the client to notify it when it was done. Unfortunately, if the client crashed, then the server memory allocated for the client might never be released.

In addition to these issues, if the process that maintained the client's session crashed, then the client's "work-in-progress" would be lost.

Why Use Web Services?

Web services make it relatively easy to reuse and share common logic with such diverse clients as mobile, desktop, and web applications. The broad reach of web services is possible because they rely on open standards that are ubiquitous, interoperable across different computing platforms, and independent of the underlying execution technologies. All web services, at the very least, use HTTP and leverage data-interchange standards like XML and JSON, and common media types. Beyond that, web services use HTTP in two distinct ways. Some use it as an application protocol to define standard service behaviors. Others simply use HTTP as a transport mechanism to convey data. Regardless, web services facilitate rapid application integration because, when compared to their predecessors, they tend to be much easier to learn and implement. Due to their inherent interoperability and simplicity, web services facilitate the creation of complex business processes through service composition. This is a practice in which compound services can be created by assembling simpler services into workflows.

Web services establish a layer of indirection that naturally insulates clients from the means used to fulfill their requests (see Figure 1.3). This makes it possible for clients and services to evolve somewhat independently as long as *breaking changes* do not occur on the service's public interface (for more on breaking changes, refer to the section What Causes Breaking Changes? in Chapter 7). A service owner may, for example, redesign a service to use an open source library rather than a custom library, all without having to alter the client.

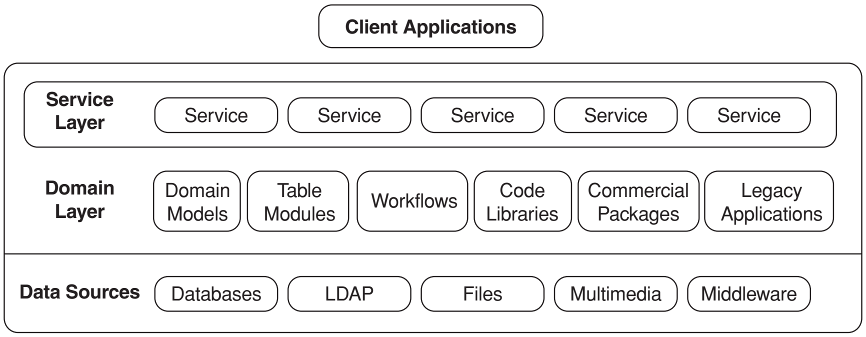

Figure 1.3 *Web services help to insulate clients from the logic used to fulfill their requests. They establish a natural layer of indirection that makes it possible for clients and domain entities (i.e., workflow logic,* Table Modules, Domain Models *[POEAA], etc.) to evolve independently.*

Web Service Considerations and Alternatives

While web services are appropriate in many scenarios, they shouldn't be used in every situation. Web services are "expensive" to call. Clients must **serialize** all input data to each web service (i.e., the request) as a stream of bytes and transmit this stream across computer processes (i.e., address spaces). The web service must **deserialize** this stream into a data format and structure it understands before executing. If the service provides a "complex type" as a response (i.e., something more than a simple HTTP status code), then the web service must serialize and transmit its response, and the client must deserialize the stream into a format and structure it understands. All of these activities take time. If the web service is located on a different machine from the client, then the time it takes to complete this work may be several orders of magnitude greater than the time required to complete a similar in-process call.

Possibly more important than the problem of latency is the fact that web service calls typically entail distributed communications. This means that client and service developers alike must be prepared to handle partial failures [Waldo, Wyant, Wollrath, Kendall]. A partial failure occurs when the client, service, or network itself fails while the others continue to function properly. Networks are inherently unreliable, and problems may arise for innumerable reasons. Connections will occasionally time out or be dropped. Servers will be overloaded from time to time, and as a result, they may not be able to receive or process all requests. Services may even crash while processing a request. Clients may crash too, in which case the service may have no way to return a response. Multiple strategies must therefore be used to detect and handle partial failures.

In light of these inherent risks, developers and architects should first explore the alternatives. In many cases, it may be better to create "service libraries" (e.g., JARs, .NET assemblies) that can be imported, called, and executed from within the client's process. If the client and service have been created for different platforms (e.g., Java, .NET), you may still use a variety of techniques that enable disparate clients and services to collaborate from within the same process. The client may, for example, be able to host the server's runtime engine, load the services into that environment, and invoke the target directly. To illustrate, a .NET client could host a Java Virtual Machine (JVM), load a Java library into the JVM, and communicate with the target classes through the Java Native Interface (JNI). You may also use third-party "bridging technologies." These options, however, can become quite complex and generally prolong the client's coupling to the service's technologies.

Web services should therefore be reserved for situations in which out-of-process and cross-machine calls "make sense." Here are a few examples of when this might occur.

- The client and service belong to different application domains and the "service functions" cannot be easily imported into the client.
- The client is a complex business process that incorporates functions from multiple application domains. The logical services are owned and managed by different organizations and change at different rates.
- The divide between the client and server is natural. The client may, for example, be a mobile or desktop application that uses common business functions.

Developers would be wise to consider alternatives to web services even when cross-machine calls seem justified.

- MOM (e.g., MSMQ, WebSphere MQ, Apache ActiveMQ, etc.) can be used to integrate applications. These technologies, however, are best reserved for use within a secured environment, far behind the corporate firewall. Furthermore, they require the adoption of an asynchronous communications style that forces all parties to tackle several new design challenges. MOM solutions often use proprietary technologies that are platform-specific. For complete coverage of this topic, see *Enterprise Integration Patterns:*

Designing, Building, and Deploying Messaging Solutions [EIP]. Web services often forward requests to MOM.

- A certain amount of overhead should be expected with HTTP due to the time it takes for clients and servers to establish connections. This added time may not be acceptable in certain high-performance/high-load scenarios. A connectionless protocol like User Datagram Protocol (UDP) can be a viable alternative for situations like these. The trade-off, however, is that data may be lost, duplicated, or received out of order.
- Most web service frameworks can be configured to stream data. This helps to minimize memory utilization on both the sender's and receiver's end because data doesn't have to be buffered. Response times are also minimized because the receiver can consume the data as it arrives rather than having to wait for the entire dataset to be transferred. However, this option is best used for the transfer of large documents or messages rather than for real-time delivery of large multimedia files like video and audio. For situations like these, protocols such as Real Time Streaming Protocol (RTSP, www.ietf.org/rfc/rfc2326.txt), Real Time Transport Protocol (RTP, http://tools.ietf.org/html/rfc3550), and Real Time Control Protocol (RTCP, http://tools.ietf.org/html/rfc3605) are usually more appropriate than HTTP.

Services and the Promise of Loose Coupling

Services are often described as being loosely coupled. However, the definitions for this term are varied and cover a broad array of concerns. Coupling is the degree to which some entity (e.g., client) depends on another entity. When the dependencies are many, the coupling is said to be high or tight (e.g., high coupling, tightly coupled). Conversely, when the dependencies are few, coupling is considered to be low or loose (e.g., low coupling, loosely coupled).

It is certainly true that web services can eliminate the client's dependencies on the underlying technologies used by a service. However, clients and services can never be completely decoupled. Some degree of coupling will always exist and is often necessary. The following list describes a few forms of coupling that service designers must consider.

- **Function coupling:** Clients expect services to consistently produce certain results given certain types of input under particular scenarios. Clients are therefore indirectly dependent on the logic implemented by web services. The client will most certainly be affected if this logic is implemented incorrectly or is changed to produce results that are not in accordance with the client's expectations.

- **Data structure coupling:** Clients must understand the data structures that a service receives and returns, the data types used in these structures, and the character encodings (e.g., Unicode) used in messages. If a data structure provides links to related services, the client must know how to parse the structure for that information. The client may also need to know what HTTP status codes the service returns. Service developers must be careful to refrain from including platform-specific data types (e.g., dates) in data structures.
- **Temporal coupling:** A high degree of temporal coupling exists when a request must be processed as soon as it's received. The implication is that the systems (e.g., databases, legacy or packaged applications, etc.) behind the service must always be operational. Temporal coupling can be reduced if the time at which a request is processed can be deferred. Web services can achieve this outcome with the *Request/Acknowledge* pattern (59). Temporal coupling is also high if the client must block and wait for a response. Clients may use the *Asynchronous Response Handler* pattern (184) to reduce this form of coupling.
- **URI coupling:** Clients are often tightly coupled to service URIs. That is, they often either have a static URI for a service, or follow a simple set of rules to construct a service URI. Unfortunately, this can make it difficult for service owners to move or rename service URIs, or to adopt new patterns for URI construction since actions like these would likely cause clients to break. The following patterns can help to reduce the client's coupling to the service's URI and location: *Linked Service* (77), *Service Connector* (168), *Registry* (220), and *Virtual Service* (222).

What about SOA?

Many definitions for Service-Oriented Architecture (SOA) have been offered. Some see it as a technical style of architecture that provides the means to integrate disparate systems and expose reusable business functions. Others, however, take a much broader view:

> *A service-oriented architecture is a style of design that guides all aspects of creating and using business services throughout their lifecycle (from conception to retirement).* [Newcomer, Lomow, p. 13]

Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. [OASIS Ref Model]

These viewpoints suggest that SOA is a design paradigm or methodology wherein "business functions" are enumerated as services, organized into logical domains, and somehow managed over their lifetimes. While SOA can help business personnel articulate their needs in a way that comes more naturally than, say, object-oriented analysis, there are still many ways to implement services. This book focuses on several technical solutions that may be used to create a SOA.

Summary

By eliminating coupling to specific computing platforms, web services have helped us overcome one of the main impediments to software reuse. However, there are many ways to go about designing services, and developers are confronted with a long list of questions that must be resolved. This book will help you find the solutions that are most appropriate for your situation.

Index