

Architecting Composite Applications and Services with TIBCO®



Paul C. Brown

FREE SAMPLE CHAPTER

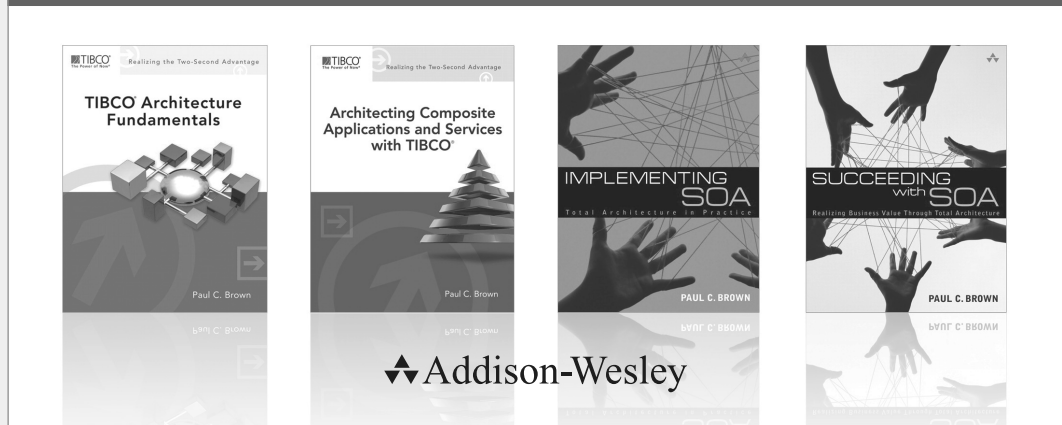


SHARE WITH OTHERS



Architecting Composite Applications and Services with TIBCO®

TIBCO® Press



Visit informit.com/tibcopress for a complete list of available publications.

TIBCO® Press provides books to help users of TIBCO technology design and build real-world solutions. The initial books – the architecture series – provide practical guidance for building solutions by combining components from TIBCO's diverse product suite. Each book in the architecture series covers an application area from three perspectives: a conceptual overview, a survey of applicable TIBCO products, and an exploration of common design challenges and TIBCO-specific design patterns for addressing them. The first book in the series, *TIBCO® Architecture Fundamentals*, addresses the basics of SOA and event-driven architectures. Each of the advanced books addresses a particular architecture style, including composite applications and services, complex event processing, business process management, and data-centric solutions.

The series emphasizes the unification of business process and system design in an approach known as *total architecture*. A technology-neutral description of this approach to distributed systems architecture is described in *Implementing SOA: Total Architecture in Practice*. Techniques for addressing the related organizational and management issues are described in *Succeeding with SOA: Realizing Business Value through Total Architecture*.



Make sure to connect with us!
informit.com/socialconnect



informIT.com
THE TRUSTED TECHNOLOGY LEARNING SOURCE

Safari
Books Online

Architecting Composite Applications and Services with TIBCO®

Paul C. Brown

◆Addison-Wesley

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

TIB, TIBCO, TIBCO Software, TIBCO Adapter, Predictive Business, Information Bus, The Power of Now, TIBCO ActiveMatrix® Adapter for Database, TIBCO ActiveMatrix® Adapter for Files (Unix/Win), TIBCO ActiveMatrix® Adapter for IBM I, TIBCO ActiveMatrix® Adapter for Kenan BP, TIBCO ActiveMatrix® Adapter for Lotus Notes, TIBCO ActiveMatrix® Adapter for PeopleSoft, TIBCO ActiveMatrix® Adapter for SAP, TIBCO ActiveMatrix® Adapter for Tuxedo, TIBCO ActiveMatrix® Adapter for WebSphere MQ, TIBCO ActiveMatrix® Administrator, TIBCO ActiveMatrix® Binding Type for Adapter, TIBCO ActiveMatrix® Binding Type for EJB, TIBCO ActiveMatrix® BPM, TIBCO ActiveMatrix BusinessWorks™, TIBCO ActiveMatrix BusinessWorks™ BPEL Extension, TIBCO ActiveMatrix BusinessWorks™ Service Engine, TIBCO ActiveMatrix® Implementation Type for C++, TIBCO ActiveMatrix® Lifecycle Governance Framework, TIBCO ActiveMatrix® Service Bus, TIBCO ActiveMatrix® Service Grid, TIBCO® Adapter for CICS, TIBCO® Adapter for Clarify, TIBCO® Adapter for COM, TIBCO® Adapter for CORBA, TIBCO® Adapter for EJB, TIBCO® Adapter for Files i5/OS, TIBCO® Adapter for Files z/OS (MVS), TIBCO® Adapter for Infranet, TIBCO® Adapter for JDE OneWorld Xe, TIBCO® Adapter for Remedy, TIBCO® Adapter SDK, TIBCO® Adapter for Siebel, TIBCO® Adapter for SWIFT, TIBCO® Adapter for Teradata, TIBCO Business Studio™, TIBCO BusinessConnect™, TIBCO BusinessEvents™, TIBCO BusinessEvents™ Data Modeling, TIBCO BusinessEvents™ Decision Manager, TIBCO BusinessEvents™ Event Stream Processing, TIBCO BusinessEvents™ Standard Edition, TIBCO BusinessEvents™ Views, TIBCO BusinessWorks™, TIBCO BusinessWorks™ BPEL Extension, TIBCO BusinessWorks™ SmartMapper, TIBCO BusinessWorks™ XA Transaction Manager, TIBCO Collaborative Information Manager™, TIBCO Enterprise Message Service™, TIBCO Enterprise Message Service™ Central Administration, TIBCO Enterprise Message Service™ OpenVMS Client, TIBCO Enterprise Message Service™ OpenVMS C Client, TIBCO® EMS Client for AS/400, TIBCO® EMS Client for i5/OS, TIBCO® EMS Client for IBM I, TIBCO® EMS Client for z/OS, TIBCO® EMS Client for z/OS (CICS), TIBCO® EMS Client for z/OS (MVS), TIBCO® EMS Transport Channel for WCF, TIBCO® General Interface, TIBCO Rendezvous®, and TIBCO Runtime Agent are either registered trademarks or trademarks of TIBCO Software Inc. and/or its affiliates in the United States and/or other countries.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Brown, Paul C.

Architecting composite applications and services with TIBCO / Paul C. Brown.

p. cm.
Includes index.

ISBN 978-0-321-80205-7 (pbk. : alk. paper) — ISBN 0-321-80205-5 (pbk. : alk. paper) 1.

Composite applications (Computer science) 2. Application software—Development. 3.

Computer network architectures. 4. TIBCO Software Inc. I. Title.

QA76.76.A65B78 2012

004.2'2—dc23

2012016968

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-80205-7

ISBN-10: 0-321-80205-5

Text printed in the United States on recycled paper at RR Donnelly in Crawfordsville, Indiana.

First printing, July 2012

*To Michael Fallon and the TIBCO Education Team:
Thank you for your perseverance teaching me
the fine art of knowledge transfer.*

Contents

Preface	xxv
Acknowledgments	xxxiii
About the Author	xxxv
Part I: Getting Started	1
Chapter 1: Components, Services, and Architectures	3
Objectives	3
Architecture Views	4
Process Model	4
Architecture Pattern	6
Process-Pattern Mapping	6
A Hierarchy of Architectures	7
Solution Architecture	8
Service or Component Specification Architecture	9
Service or Component Implementation Architecture	9
Why Make These Architecture Distinctions?	11
Solutions Comprising Dedicated Components	11
Solutions Comprising Shared Services	12
Design Patterns: Reference Architectures	13
Solution Architecture	14
Solution Architecture Pattern	14
Solution Business Processes	14
Solution Process Mapping	14

Service Architecture	17
Service Utilization Pattern	17
Service Utilization Architecture Pattern	18
Service Utilization Process Models	18
Service Utilization Process Mappings	19
Composite Service Architecture	20
Composite Service Architecture Pattern	20
Composite Service Process Mapping	21
Service Utilization Contract	22
Component Life Cycle	22
Summary	23
Chapter 2: <i>TIBCO® Architecture Fundamentals Review</i>	25
Objectives	25
Products Covered in <i>TIBCO® Architecture Fundamentals</i>	25
TIBCO Enterprise Message Service™	26
TIBCO ActiveMatrix® Product Portfolio	27
ActiveMatrix® Service Bus	27
ActiveMatrix® Service Grid	28
ActiveMatrix BusinessWorks™	28
ActiveMatrix® Adapters	30
ActiveMatrix Deployment Options	31
Design Patterns	33
Basic Interaction Patterns	33
Event-Driven Interaction Patterns	35
Mediation Patterns	36
External System Access Patterns	37
Coordination Patterns	40
ActiveMatrix Service Bus Policies	44
Summary	46

Chapter 3: TIBCO Products	47
Objectives	47
Hawk®	48
Overview	48
Hawk Agent	49
Hawk Rules	49
Hawk Microagent Adapter (HMA)	53
Microagent Interfaces	54
Hawk Display	55
Hawk Event Service	55
Hawk Adapters	56
TIBCO® Managed File Transfer Product Portfolio	56
Mainframe and iSeries Integration	58
Mainframe and iSeries Interaction Options	58
Interaction Intent	59
Substation ES	60
TIBCO Mainframe Service Tracker	60
TIBCO ActiveMatrix BusinessWorks Plug-in for CICS	61
TIBCO Adapter for IBM i	61
Mainframe and iSeries File Adapters	62
BusinessConnect™	63
TIBCO Collaborative Information Manager	64
Summary	65
Chapter 4: Case Study: Nouveau Health Care	67
Objectives	67
Nouveau Health Care Solution Architecture	68
Nouveau Health Care Business Processes	68
Nouveau Health Care Architecture Pattern	70
Nouveau Health Care in Context	71
Processing Claims from Providers	71

Payment Manager Service Specification	73
Payment Manager Specification: Process Overview	74
Health Care Claim	74
Manage Payments Processes	75
Process Coordination	76
Payment Manager Specification: Domain Model	77
Accounts and Funds Transfers	77
Settlement Accounts	78
Settlement Concepts	79
Payment Domain Concepts	80
Payment Manager Specification: Interfaces	82
Payment Manager Specification: Processes	83
Immediate Payment Process	83
Deferred Payment and Settlement Processes	85
Summary	91
Part II: Designing Services	93
Chapter 5: Observable Dependencies and Behaviors	95
Objectives	95
The Black Box Perspective	96
Facets of Observable Dependencies and Behaviors	97
Example: Sales Order Service	97
Placing the Order	98
Order Shipped	102
Order Delivered	104
Observable State Information	104
Observable State and Cached Information	108
Avoiding Caches: Nested Retrieval	111
Characterizing Observable Dependencies and Behaviors	111
Context	111
Usage Scenarios	112
Triggered Behaviors	113

Observable State	114
Coordination	114
Constraints	115
Nonfunctional Behavior	116
Some Composites May Not Be Suitable for Black Box Characterization	117
Summary	119
Chapter 6: Service-Related Documentation	121
Objectives	121
Service One-Line Description and Abstract	122
Service One-Line Description	122
Service Abstract	122
Service Specification Contents	123
One-Line Description	124
Abstract	124
Context	124
Intended Utilization Scenarios	125
Interface Definitions	125
References	125
Observable State	125
Triggered Behaviors	126
Coordination	126
Constraints	127
Nonfunctional Behavior	127
Deployment Specifics	127
Example Service Specification: Payment Manager	128
Service One-Line Description	128
Service Abstract	128
Service Context	128
Intended Utilization Scenarios	129
Payment Manager Service Interfaces	129
Referenced Interfaces	132
Observable State	134

Triggered Behaviors	136
Coordination	138
Constraints	140
Nonfunctional Behavior	140
Deployment Specifics	141
Service Usage Contracts	142
Organizations	142
Service Consumers and Access Mechanisms	143
Functional Requirements	143
Nonfunctional Requirements	144
Deployment	144
Service Architecture	144
Payment Manager Architecture Pattern	145
Payment Manager Behavior Implementations	145
Payment Manager Behaviors	145
Summary	149
Chapter 7: Versioning	151
Objectives	151
Dependencies and Compatibility	152
Packages	152
OSGI Versioning	153
The Version Numbering Scheme	153
Expressing Dependencies as Ranges	154
Versioning and Bug Fixes	155
WSDL and XML Schema Versioning	156
WSDL Scope	157
XML Schema Scope	158
Version Number Placement for WSDLs and XML Schemas	159
Version Numbers in Filenames	159
Version Numbers in Namespace Names	159
Version Numbers in soapAction Names	160

Backwards-Compatible WSDL and XML	
Schema Changes	160
Adding Definitions to a WSDL	160
Deleting Unused Definitions from a WSDL (Conditional)	161
Adding Definitions to an XML Schema	161
Deleting Unused Definitions from an XML Schema (Conditional)	161
Replacing a SimpleType with an Identical SimpleType	161
Adding an Optional Field to an XML Schema (Conditional Future)	162
Incompatible Changes	163
Incompatible Changes in a Schema	163
Incompatible Changes in a WSDL	163
Rules for Versioning WSDLs and Schemas	164
Architecture Patterns for Versioning	165
Accommodating a Compatible Change	165
Incompatible Change Deployment	166
Versioning SOAP Interface Addresses (Endpoints)	168
Versioning the SOAP Action	168
How Many Versions Should Be Maintained?	169
Summary	171
Chapter 8: Naming Standards	173
Objectives	173
Using This Chapter	174
Concepts	174
Abstract Services	174
WSDL Interface Definitions	176
Relating Abstract and WSDL-Defined Services	178
Why Are Names Important?	180
Names Are Difficult to Change	180
Name Structures Define Search Strategies	181
Name Structures Define Routing Strategies	181

What Needs a Name?	182
Structured Name Design Principles	183
Use a General-to-Specific Structure	183
Employ Hierarchical Naming Authorities	185
Base Naming on Stable Concepts	187
Avoid Acronyms and Abbreviations	188
Distinguish Types from Instances	189
Plan for Multi-Word Fields	189
Use a Distinct WSDL Namespace URI for Each Interface	190
Incorporate Interface Major Version Numbers	191
Applying Naming Principles	191
Idealized Name Structures	191
Functional Context	192
WSDL and XSD Namespace URIs	194
WSDL and XSD Filenames	195
WSDL Names	196
Schema Locations	197
WSDL-Specific Schema Location	197
WSDL Message Names	197
Port Type, Service, and Binding Names	198
Operation Names	200
SOAP Address Location	200
soapAction Names	203
Schema Types Specific to an Interface and an Operation	203
Schema Shared Data Types (Common Data Model Types)	204
Complicating Realities	205
Technology Constraints	205
Naming Authorities	206
Complex Organizational Structures	206
Environments: Values for Address Location Variables	209
Deployment Flexibility for HA and DR	210
Developing Your Standard	211
Summary	212

Chapter 9: Data Structures	215
Objectives	215
Domain Models	215
Information Models	218
Data Structure Design	220
Deep Data Structures	220
Flattened Data Structures	222
Shallow Data Structures	223
Reusability	224
Common Data Models	224
Representing an Entity	225
Representing an Association	226
Designing an XML Schema	227
Identify Required Entities and Associations	228
Determine Entity Representations	229
Create the Information Model	229
Create the Schema	232
Organizing Schema and Interfaces	233
Example Schema	235
Summary	235
 Part III: Service Architecture Patterns	 237
Chapter 10: Building-Block Design Patterns	239
Objectives	239
Solution Architecture Decisions	240
Separating Interface and Business Logic	240
Design Pattern: Separate Interface and Business Logic	241
Using Services for Accessing Back-End Systems	243
Rule Service Governing Process Flow	244
Design Pattern: Rule Service Separated from Process Manager	245
Design Pattern: Rule Service as Process Manager	246

Hybrid Rule-Process Approaches	247
Design Pattern: Rule Service Assembles Process Definition	248
Design Pattern: Rule Service Directs Process Manager	249
Rule Services and Data	250
Design Pattern: Rule Client Provides Data	250
Design Pattern: Rule Service Acquires Data	250
Business Exceptions: Services Returning Variant Business Responses	252
Asynchronous JMS Request-Reply Interactions	257
The Problem with Temporary Destinations	257
Asynchronous Request-Reply: Single Reply Destination	258
Asynchronous Request-Reply: Multiple Reply Destinations	260
Supporting Dual Coordination Patterns	261
Summary	262
Chapter 11: Load Distribution and Sequencing Patterns	265
Objectives	265
Using IP Redirectors to Distribute Load	266
Using JMS Queues to Distribute Load	266
Partitioning JMS Message Load between Servers	267
Planning for Future Partitioning	267
Partitioning by Nature of Traffic	268
A Word of Caution Regarding Over-Partitioning	269
Enterprise Message Service Client Connection Load Distribution	269
Client Load Distribution: Topic Pattern	269
Client Load Distribution: Queue Pattern	270
Client Load Distribution: Combined Patterns	271
Load Distribution in ActiveMatrix Service Bus	271
ActiveMatrix Service Bus Load Distribution Using Virtual Bindings	272
ActiveMatrix Service Bus Load Distribution of Promoted Services	272

The Sequencing Problem	273
Sequencing	274
Limited Sequencing: Partial Ordering	274
Patterns That Preserve Total Sequencing	275
Single-Threaded Pattern	275
Sequence Manager Pattern	276
Load Distribution Patterns That Preserve	
Partial Ordering	278
Preserving Partial Order Sequencing in ActiveMatrix	
BusinessWorks	278
Two-Tier Load Distribution Preserving Partial Ordering	279
Implementing Two-Tier Load Distribution	
Preserving Partial Ordering	279
Summary	280
Chapter 12: Data Management Patterns	283
Objectives	283
System-of-Record Pattern	284
System of Record with Cached Read-Only Copies Pattern	285
Replicated Data with Transactional Update Pattern	286
Edit-Anywhere-Reconcile-Later Pattern	287
Master-Data-Management Pattern	288
Summary	290
Chapter 13: Composites	293
Objectives	293
What Is a Composite?	293
Specifying a Composite	294
Architecting a Composite	294
Composite Architecture Pattern	295
Composite Behavior Management: Orchestration	
and Choreography	296
Composite Processes	298
Composite Mappings	299

Completing the Component Specifications	303
The Process of Architecting a Composite	303
Composite Services and Applications	303
Information Retrieval Design Patterns	304
Cascading Control Pattern	304
Cached Information Pattern	304
Lookup and Cross Reference	306
TIBCO ActiveMatrix Composite Implementation	307
Defining the Composite	307
Selecting Implementation Types	307
Summary	308
 Part IV: Advanced Topics	 311
Chapter 14: Benchmarking	313
Objectives	313
Misleading Results	314
The Bathtub Test	314
Disabling Features	315
Test Harness Design	316
Determining Operating Capacity	316
Documenting the Test Design	317
Test Harness Architecture Pattern	317
Test Harness Process Mapping	318
Experimental Parameters	321
Test Results	322
Benchmarking Complex Components	324
Overhead Benchmarks	325
Individual Activity Benchmarks	326
Benchmark Common Scenarios in which Activities Interact	326
Interpreting Benchmark Results	327
Identifying the Capacity Limit	327
Identifying CPU Utilization Limits	328

Identifying Network Bandwidth Limits	329
Identifying Disk Performance Limits	331
Identifying Memory Limits	335
Identifying Test Harness Limits	336
Using Benchmark Results	336
Summary	338
Chapter 15: Tuning	341
Objectives	341
ActiveMatrix Service Bus Node Architecture	341
Thread Pools	342
Worker Thread Assignments	345
Default SOAP/HTTP Thread Usage	345
Default JMS Thread Usage	347
Default Node-to-Node Communications Thread Usage	349
Thread Usage with the Virtualize Policy Set	351
Thread Usage with the Threading Policy Set	353
ActiveMatrix BusinessWorks™ Service Engine Architecture	357
Thread Pools	357
ActiveMatrix BusinessWorks Internal Architecture	358
JMS Process Starters	360
Starting Jobs from the ActiveMatrix Service Bus	361
ActiveMatrix BusinessWorks Job Processing	362
ActiveMatrix BusinessWorks Tuning Parameters	365
Summary	369
Chapter 16: Fault Tolerance and High Availability	371
Objectives	371
Common Terms	372
Deferred JMS Acknowledgement Pattern	373
Intra-Site Cluster Failover Pattern	374
Generic Site Failover	377

Storage Replication Strategies and the Recovery Point Objective	377
Inter-Site Failover with Different Host Identity Pattern	378
Inter-Site Failover with Same Host Identity Pattern	380
Enterprise Message Service Failover	381
EMS File-Based Intra-Site Failover Pattern	382
EMS Database-based Intra-Site Failover Pattern	383
EMS Client Configuration for Failover	384
Considerations in Selecting an EMS Failover Strategy	384
ActiveMatrix BusinessWorks Failover	385
ActiveMatrix BusinessWorks Deferred JMS Acknowledgement Pattern	386
ActiveMatrix BusinessWorks Built-In Intra-Site Failover Pattern	387
ActiveMatrix BusinessWorks Cluster Failover Pattern	388
ActiveMatrix Service Bus Failover	390
Using the Deferred JMS Acknowledgement Pattern	390
Using the ActiveMatrix BusinessWorks Built-In Intra-Site Failover Pattern	390
Using the Intra-Site Cluster Failover and Inter-Site Failover with Same-Host Identity Patterns	391
An Example of a 99.999% Availability Environment for the Enterprise Message Service	391
EMS Multi-Site with No Message Persistence Pattern	391
EMS Multi-Site with Message Persistence Pattern	393
Summary	396
Chapter 17: Service Federation	401
Objectives	401
Factors Leading to Federation	402
Issues in Federation	402
Access Control	402
Repositories	403
Basic Federation Pattern	403

Federation with Remote Domain Pattern	405
Distributed Federation Pattern	406
Standardizing Service Domain Technology	407
Summary	407
Chapter 18: Documenting a Solution Architecture	409
Business Objectives and Constraints	409
Quantified Business Expectations	409
Business Constraints	410
Business Risks	410
Solution Context	410
Business Process Inventory	410
Domain Model	410
Solution Architecture Pattern	411
Business Process 1	411
Business Process Design	411
Process-Pattern Mapping	411
Business Process 2	411
Business Process <i>n</i>	412
Addressing Nonfunctional Solution Requirements	412
Performance and Scalability	412
Availability within a Data Center	412
Site Disaster Recovery	412
Security	412
Component/Service A	413
Business Process Involvement	413
Interfaces	414
Observable Architecture	414
Observable State	414
Coordination	414
Constraints	414
Nonfunctional Behavior	414
Component/Service B	415

Component/Service n	415
Deployment	416
Deployment Environment Migration	416
Development Configuration	416
Test Configuration	416
Production Configuration	416
Integration and Testing Requirements	416
Integration Strategy	416
Behavioral Testing	416
Failure Testing	417
Performance Testing	417
Appendix A: Common Data Format Specifications	417
Appendix B: Message Format Specifications	417
Appendix C: Service Interface Specifications	417
Appendix D: Data Storage Specifications	417
Chapter 19: Documenting a Service Specification	419
Service Overview	419
Service One-Line Description	419
Service Abstract	420
Service Context	420
Intended Utilization Scenarios	420
Interface Definitions	421
Referenced Components	421
Observable State	421
Triggered Behaviors	422
Coordination	422
Constraints	422
Nonfunctional Behavior	422
Deployment	423
Appendix A: Service Interface Specifications	423
Appendix B: Referenced Interface Specifications	423

Afterword	425
Appendix A: UML Notation Reference	427
Class Diagram Basics	427
Classes	427
Interfaces	428
Associations	429
Part-Whole Relationships: Aggregation and Composition	430
Generalization	431
Structure	432
Composite Structure Diagrams	432
Avoiding Common Structure Editing Mistakes	434
Execution Environments	435
Execution Environment Structure	435
Ports	436
Activity Diagrams	437
Actions and Control Flow	437
Artifacts	437
Structured Activities	438
Call Operation Action	438
Decisions, Forks, and Joins	438
Event Actions	439
Swimlanes	439
Swimlanes with Structure	440
Collaborations	440
State Machines	441
States and Transitions	441
Composite States	441
Orthogonal States	441

Appendix B: WSDLs and Schemas from Examples	443
Sales Order Example	443
Sales Order Service Interface WSDL	443
Address Schema	444
Carrier Schema	445
Customer Schema	446
Manufacturer Schema	447
Phone Schema	448
Product Schema	448
Order Fulfillment Schema	449
Sales Order Interface Schema	450
Sales Order Schema	451
Index	453

Preface

About This Book

In composite applications and services, multiple components collaborate to provide the required functionality. These are distributed solutions for which there are many possible architectures. Some of these will serve the enterprise well, while others will lead to dead-end projects.

The Dual Roles of an Architecture

At the core lies an understanding of the dual role that architecture plays in a distributed design. One role is as an expression of an overall design: how the components collaborate to solve the problem. The other role is as a specification for the components and services that are part of that design. Understanding this dual role leads to an understanding of the information that must be present in an architecture document in order to effectively play this dual role.

This dual perspective on architecture is recursive: Dive in to a component or service and you'll find that it, too, has an architecture that must play both of these roles. It describes how its sub-components collaborate to provide the capabilities of the component. At the same time, it serves as a specification for each of its sub-components. And so on.

This perspective is so important to success that the better parts of three chapters (1, 5, and 6) are devoted to its exploration both by discussion and by example. All of the examples in the book are documented in the same manner.

Design Patterns

The architectures you define provide solutions for your enterprise. Many of these require solutions to well-known problems. To aid you in this work, this book covers a wide variety of design patterns for addressing

common challenges. These include partitioning interfaces and business logic; incorporating rules into your design; asynchronous interactions involving multiple consumers and providers; providing services with both synchronous and asynchronous coordination patterns; distributing workload with and without sequencing constraints; managing replicated data; creating composite components, services, and applications; fault-tolerance, high-availability, and site disaster recovery; and federating services. These build upon the basic service and integration patterns covered in the book *TIBCO® Architecture Fundamentals*.¹

Relevant TIBCO Products

Components and services need to be implemented, of course, and to do this appropriate technologies need to be selected. TIBCO provides a number of products that are intended to play specific roles in composite applications and services. This book provides an overview of these roles for TIBCO ActiveMatrix® Service Bus, TIBCO ActiveMatrix® Service Grid, TIBCO ActiveMatrix BusinessWorks™, TIBCO Hawk®, TIBCO® Managed File Transfer components, TIBCO® mainframe integration products, TIBCO BusinessConnect™, and TIBCO Collaborative Information Manager™.

Best Practices

A good architecture is a living thing that evolves gracefully over time as the demands facing the enterprise change. Facilitating this evolution requires careful consideration of service and data structure versioning, naming standards, modular data structure design, and the federation of services. This book devotes a chapter to each of these topics.

Performance and Tuning

Your solution must perform well enough to meet the needs of the enterprise. Assuring yourself that you have achieved the performance goals requires suitable benchmarking, and one chapter is devoted to the conduct of such experiments and the interpretation of results. Hand in hand with benchmarking goes tuning, and one chapter is

1. Paul C. Brown, *TIBCO® Architecture Fundamentals*, Boston: Addison-Wesley (2011).

devoted to the tuning of ActiveMatrix® Service Bus and ActiveMatrix BusinessWorks™.

Service Federation

Services will arise in many contexts both within and external to your enterprise. Some of these services will be intended for local use, while others will be intended for more widespread use. One chapter is devoted to service federation, which focuses on organizing services that arise in these different contexts.

Documenting Solution Architectures and Service Specifications

Finally, all of your architecture decisions must be captured in a form that can be communicated to all interested parties, from the business people who chartered the project to the technical teams that will implement, deploy, and operate the components that comprise the finished solution. One chapter is devoted to documenting solution architectures and another to documenting service specifications. Augmenting these chapters are online templates for each of these documents and a worked example of each. These may be found at informit.com/title/9780321802057.

In summary, this book is a guide to successfully architecting composite applications and services employing TIBCO technologies. It presents a comprehensive approach for architecting everything from the overall solution to the individual components and services. It builds upon and extends the basic design patterns and product information presented in the book *TIBCO® Architecture Fundamentals*.

TIBCO Architecture Book Series

Architecting Composite Applications and Services with TIBCO® is the second book in a series on architecting solutions with TIBCO products (Figure P-1). It builds upon the material covered in *TIBCO® Architecture Fundamentals*, which provides material common to all TIBCO-based designs. Each of the more advanced books, including this one, explores a different style of solution, all based on TIBCO technology. Each explores the additional TIBCO products that are relevant to that style of

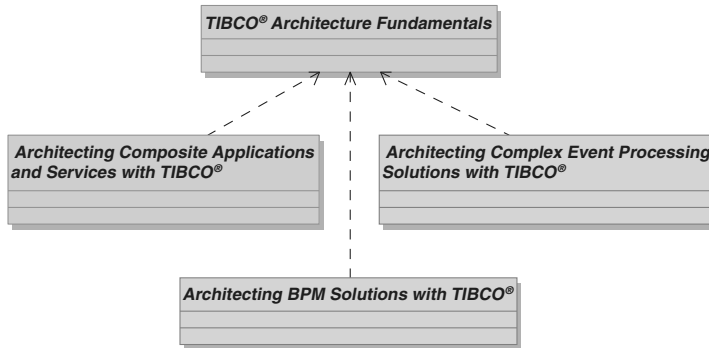


Figure P-1: *TIBCO Architecture Book Series*

solution. Each defines larger and more specialized architecture patterns relevant to the style, all built on top of the foundational set of design patterns presented in *TIBCO® Architecture Fundamentals*.

Intended Audience

Project architects are the intended primary audience for this book. These are the individuals responsible for defining an overall solution and specifying the components and services required to support that solution. Experienced architects will find much of interest, but no specific prior knowledge of architecture is assumed in the writing. This is to ensure that the material is also accessible to novice architects and advanced designers. For this latter audience, however, a reading of *TIBCO® Architecture Fundamentals* is highly recommended. It explores basic concepts in greater detail. This book provides a summary of that material in Chapter 2.

TIBCO specialists in a center of excellence will find material of interest, including background on the TIBCO product stack and design patterns showing best-practice uses of TIBCO products. The material on benchmark testing and tuning ActiveMatrix Service Bus and ActiveMatrix BusinessWorks lay the foundations for building high-performance applications based on these products.

Enterprise architects will find content of interest as well. The material on architecture documentation, component and service specification, versioning, namespace design, modular data structure design,

and benchmarking can be used as a reference for defining or reviewing enterprise standards in these areas. The collection of design patterns, in conjunction with those presented in *TIBCO® Architecture Fundamentals*, provide the basis for a baseline set of standard design patterns for the enterprise.

Technical managers will also find material of interest, particularly the description of the content expected in architecture documents and specifications. The guidelines for conducting benchmark tests will also be of interest.

Detailed Learning Objectives

After reading this book, you will be able to

- Create and document architectures for solutions, component and service specifications, and component and service implementations.
- Describe the intended roles for ActiveMatrix Service Bus, ActiveMatrix® Service Grid, ActiveMatrix BusinessWorks, Hawk®, TIBCO Managed File Transfer components, TIBCO mainframe integration products, BusinessConnect, and TIBCO Collaborative Information Manager in composite applications and services.
- Define a manageable approach to versioning services and their artifacts.
- Establish practical standards for naming services and related artifacts.
- Design modular and manageable data structures.
- Conduct and interpret performance benchmark tests.
- Tune ActiveMatrix Service Bus and ActiveMatrix BusinessWorks™.
- Identify and select appropriate design patterns for
 - Separating interface and business logic
 - Incorporating rules into an architecture
 - Supporting asynchronous interactions involving multiple consumers and providers
 - Simultaneously supporting synchronous and asynchronous coordination patterns
 - Distributing workload with and without sequencing constraints

- Managing replicated data
- Creating composites of components and services
- Fault-tolerance, high-availability, and site disaster recovery
- Federating services

Organization of the Book

This book is divided into four parts (Figure P-2). Part I begins with a discussion of components, services, and architectures that provides the conceptual foundation for the book. It next reviews the material covered in *TIBCO® Architecture Fundamentals*, which is foundational material for this book. Next comes a discussion of some TIBCO products not covered in *TIBCO® Architecture Fundamentals* that play prominent roles in composite applications and services. Finally, the Nouveau Health Care case study is introduced. This rich case study will be used as the basis for many of the examples in the book.

Part II covers the basics of designing services. It starts with a discussion of observable dependencies and behaviors—the externally observable characteristics of a component or service. This understanding is next incorporated into a discussion of service-related documentation. The following three chapters address issues that greatly impact the flexibility of the architecture: versioning, naming standards, and data structures.

Part III describes a number of common design challenges and architectural patterns that can be used to address them. It starts with a number of general building-block patterns. Next are patterns for load distribution, with and without sequencing constraints. Patterns for managing replicated data are then followed by patterns for composing components and services.

Part IV addresses advanced topics. The conduct and interpretation of benchmark experiments are key for achieving performance goals, as is the tuning of key products. Fault-tolerance, high-availability, and site disaster recovery are discussed, followed by a discussion about federating multiple service domains. The book concludes with chapters covering the documentation of a solution and a service specification.

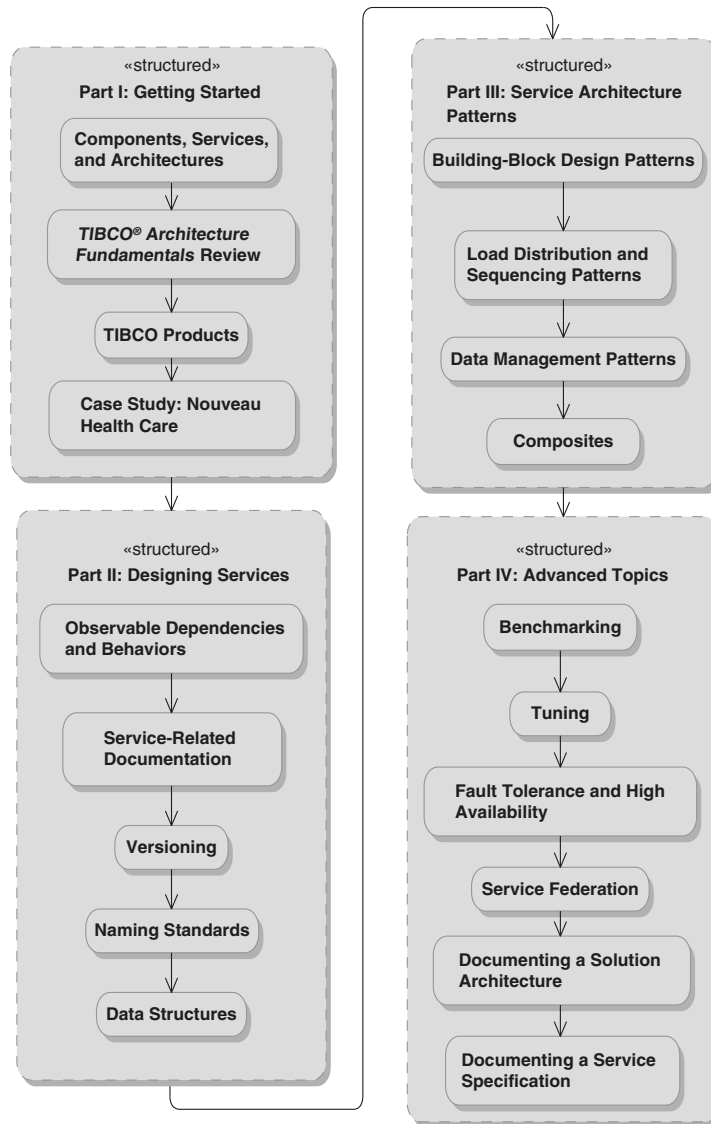


Figure P-2: *Organization of the Book*

Acknowledgments

Knowingly or unknowingly, many people have contributed to this book. Chief among these are my fellow global architects at TIBCO, particularly Kevin Bailey, Todd Bowman, Richard Flather, Ben Gundry, Nochum Klein, Marco Malva, Heejoon Park, and Michael Roeschter. Many of the other members of the architectural services team have contributed as well, including JenVay Chong, Dave Compton, Roger Kohler, Ed Presutti, and Michael Zhou. My thanks to these folks and the rest of the architectural services group—you're a great team to work with.

The educational services team at TIBCO has played a key role in conceptualizing this book, along with the accompanying course and architect certification program. More broadly, they have taught me the fine art of knowledge transfer and greatly influenced the manner in which this material is presented. Particular thanks to Alan Brown, Michael Fallon, Mike Goldsberry, Robert Irwin, Michelle Jackson, Lee Kleir, Madan Mashalkar, Tom Metzler, Howard Okrent, and Ademola Olateju.

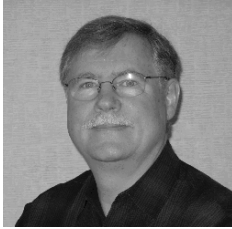
The folks in TIBCO engineering have been more than generous with their time helping me understand the architecture underlying their products, including Bill Brown, John Driver, Michael Hwang, Eric Johnson, Collin Jow, Salil Kulkarni, Erik Lundevall, Bill McLane, Jean-Noel Moyne, Kevin O'Keefe, Denny Page, and Shivajee Samdarshi. My specific thanks to those who corrected my flawed understanding and provided the foundation for the threading diagrams in this book: Mohamed Aariff, Praveen Balaji, Seema Deshmukh, Laurent Domenech, Tejaswini Hiwale, Sabin Ielceanu, Shashank Jahagirdar, Salil Kulkarni, and Pankaj Tolani.

I wish to thank the management team at TIBCO whose support made this book possible: Paul Asmar, Eugene Coleman, Jan Plutzer, Bill Poteat, Murray Rode, and Murat Sonmez.

I would like to thank those who took the time to review the manuscript for this book and provide valuable feedback: Rodrigo Candido de Abreu, Sunil Apte, Abbey Brown, Antonio Bruno, Jose Estefania, Marcel Grauwen, Ben Gundry, Nochum Klein, Lee Kleir, Alexandre Jeong, Michael Roeschter, Peter Schindler, Mohamed Shahat, Mark Shelton, Mohan Sidda, and Nikky Sooriakumar.

Finally, I would like to once again thank my wife, Maria, who makes all things possible for me.

About the Author



Dr. Paul C. Brown is a Principal Software Architect at TIBCO Software Inc., and is the author of *Succeeding with SOA: Realizing Business Value Through Total Architecture* (2007), *Implementing SOA: Total Architecture in Practice* (2008), and *TIBCO® Architecture Fundamentals* (2011), all from Addison-Wesley. He is also a co-author of the SOA Manifesto (soa-manifesto.org). His model-

based tool architectures are the foundation of a diverse family of applications that design distributed control systems, process control interfaces, internal combustion engines, and NASA satellite missions. Dr. Brown's extensive design work on enterprise-scale information systems led him to develop the total architecture concept: business processes and information systems are so intertwined that they must be architected together. Dr. Brown received his Ph.D. in computer science from Rensselaer Polytechnic Institute and his BSEE from Union College. He is a member of IEEE and ACM.

Observable Dependencies and Behaviors

Objectives

When you are creating a design as a collection of interacting components (e.g., services), it is useful to be able to ignore the internals of those components and concentrate on their interactions. However, you still need to know something about the component—how it depends upon the environment in which it operates and how it will behave. This chapter is about how to characterize these dependencies and behaviors. As we shall see in the next chapter, this type of characterization forms the core of component and service specifications.

If you are conceptualizing (defining) the component as part of your design, you will be called upon to create this characterization. If you are using an existing component, then you will be the consumer of the characterization. Either way, you need to understand what is required to appropriately characterize the component.

Behavior is the way in which something, in our case a component, responds to a stimulus. To effectively utilize the component in a solution,

you need to understand how the component will respond to stimuli provided by the other solution components and what stimuli the component will provide to the rest of the solution.

In this work we will use the term *observable dependency* to refer to the relationship between the component and components in the environment upon which it depends. We will use the term *observable behavior* to refer to the behavior as seen by the rest of the solution—without looking inside the component. After reading this chapter you should be able to describe the concepts of observable dependency and observable behavior and explain how they can be characterized. This will provide the foundation for an ensuing discussion of component and service specifications.

The Black Box Perspective

When you have a solution comprised of collaborating components (services), it is easier to understand the solution if you are able to view each component as a black box and not have to worry about what is inside. To do this, you need to be able to characterize two things: (1) the observable dependencies of the black box—the components in its environment upon which it depends, including their interfaces and behaviors, and (2) the observable behavior of the black box—how it responds to stimuli provided by other solution components and which stimuli it provides to those components. That’s what this chapter is about.

There is a strong analogy here to the way in which people learn about things in their environment. When presented with an object they want to learn about, people are inclined to pick it up, turn it at different angles, and experiment with different ways in which they can interact with it. On electronic gadgets they press the keys, flip the switches, and touch the screen, all the while observing both the dependencies and the resultant behavior. They are making observations about the object.

You can think about solution components the same way, only the observations you make are not visual. The other components upon which they rely for proper operation characterize their dependencies, and the stimuli are the invocations of interface operations and other events that trigger component responses. Observations are the responses, such as returned data structures and the invocations of other component’s interfaces. What you learn about the component are its dependencies and observable behavior.

Of course, experimenting with a component to learn its dependencies and behaviors is likely to be a time-consuming and error-prone activity. Preferably you would like to have someone tell you how the component will behave and the components upon which it depends for proper operation. Such a characterization of dependency and behavior forms the core of the component's specification.

Facets of Observable Dependencies and Behaviors

So what do you need to know about a component to describe its observable dependencies and behaviors? Some important facets include

- External component dependencies upon the component as characterized by the component's interfaces and operations
- The component's dependencies upon external components, consisting of the identification of these components and the characterization of their interfaces and operations
- Usage scenarios: characterizations of the business processes in which the component is expected to participate and the component's participation in those processes
- Triggered behaviors: the structure of component activities that explain the relationships between the component's triggers, responses, inputs, observable state, and outputs
- Observable state: information retained by the component with a presence that is observable through its interactions
- Coordination: the manner in which the component's activity can be coordinated with that of external components
- Constraints: limitations, particularly on the sequencing of triggers
- Nonfunctional behavior: performance, availability, and so on

Example: Sales Order Service

We will use the example of a Sales Order Service supporting an Order-to-Delivery business process to illustrate the various facets of observable dependencies and behaviors. An overview of the Order-to-Delivery business process is depicted in Figure 5-1. This figure presents a usage scenario for the Sales Order Service, which is one of the participants in

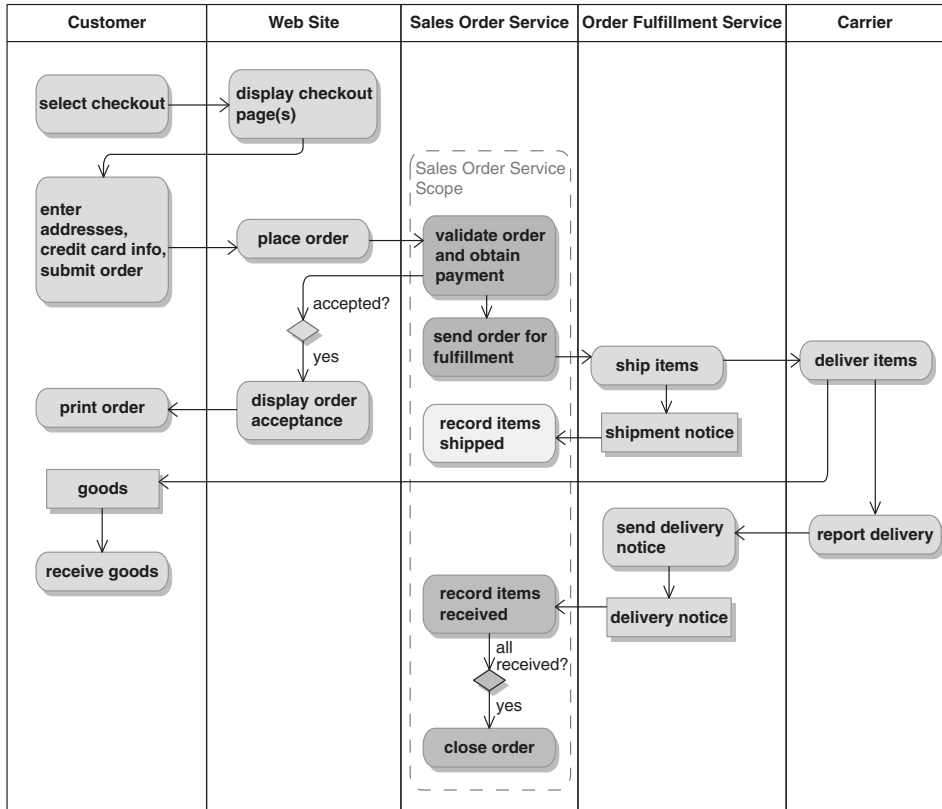


Figure 5-1: Sales Order Service in the Order-to-Delivery Business Process

the process. The Sales Order Service manages the full life cycle of an order. Let's examine how it participates in the process and note the relevant observable facets that come into play.

Placing the Order

The Sales Order Service accepts an order from the Web Site via the `placeOrder()` operation of the Sales Order Service Interface (Figure 5-2). The invocation of this operation constitutes a trigger, and the fact that it is invoked by the Web Site indicates a dependency on the interface.

The invocation of the `placeOrder()` operation triggers an associated behavior of the service. This behavior validates the order and obtains payment, in the process deciding whether or not to accept the

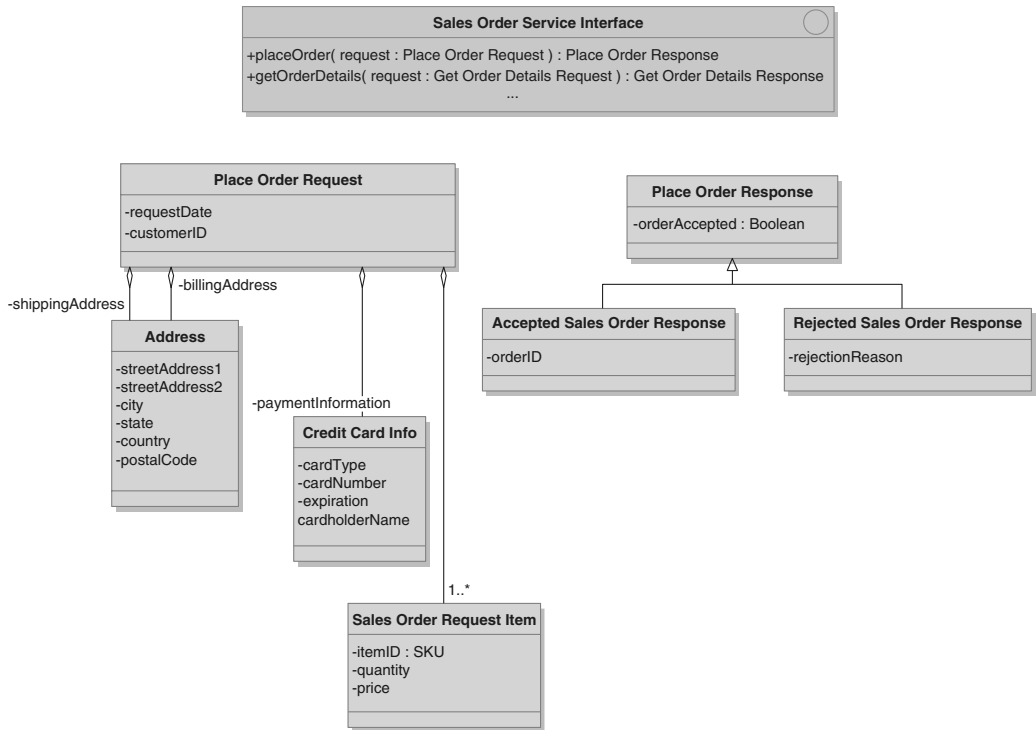


Figure 5-2: *Sales Order Service Interface—placeOrder () Operation*

order. These activities involve interactions with other components, components that are not part of the service and are thus dependencies. Let's look at the interfaces involved, and later we'll identify the components that provide those interfaces.

The validity of each `itemID` is established by calling the `validate-ProductID ()` operation of the Product Query Interface (Figure 5-3).

The validity of the `customerID` is established by calling the `get-Customer ()` operation of the Customer Query Interface (Figure 5-4).

Payment is obtained by calling the `obtainPayment ()` operation on the Credit Interface (Figure 5-5).

At this point the service returns the Place Order Response to the waiting caller of `placeOrder ()`. This data structure indicates whether the order was accepted and, if not accepted, the reason why. The data structure exposes the fact that the service is validating the order and obtaining payment, thus making this portion of the behavior observable to the caller of `placeOrder ()`.

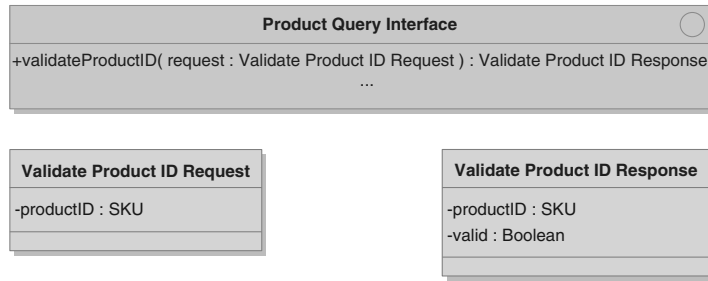


Figure 5-3: *Product Query Interface—validateProductID() Operation*

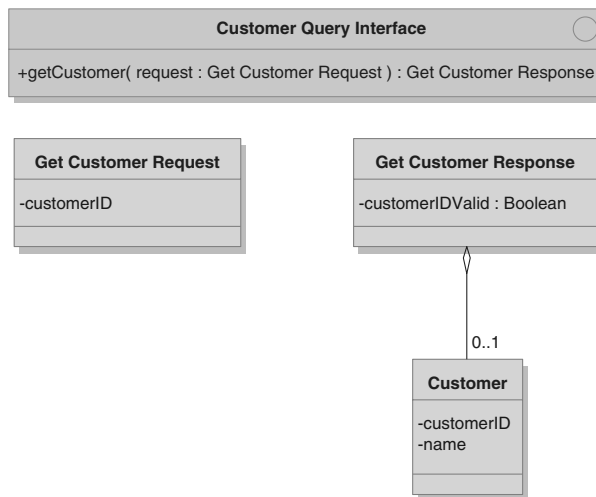


Figure 5-4: *Customer Query Interface—getCustomer() Operation*

The observable behavior does not end with the return of the Place Order Response data structure. If the order is accepted, the service then sends the order to Order Fulfillment using the `fillOrder()` operation of the Order Fulfillment Interface to accomplish this (Figure 5-6). This is yet another dependency. Note that, as designed, this is an In-Only operation and does not return a response.

At this point the behavior that began with the invocation of the `placeOrder()` operation comes to a conclusion. Putting all the pieces together, the behavior triggered by this invocation is that shown in Figure 5-7. This diagram also indicates components for which there are observable dependencies: Product Service, Customer Service, Credit Service, and Order Fulfillment Service.

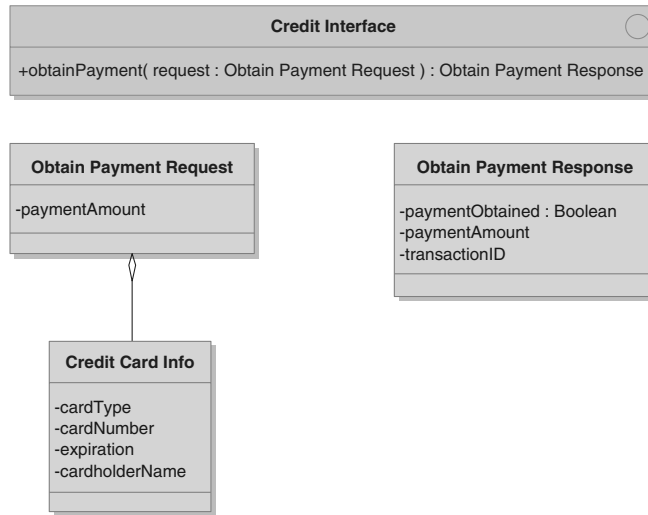


Figure 5-5: *Credit Interface—obtainPayment () Operation*

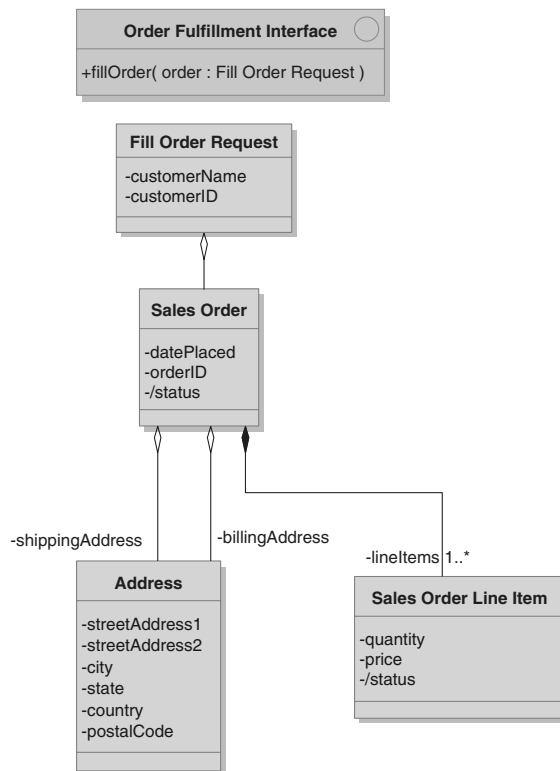


Figure 5-6: *Order Fulfillment Interface—fillOrder () Operation*

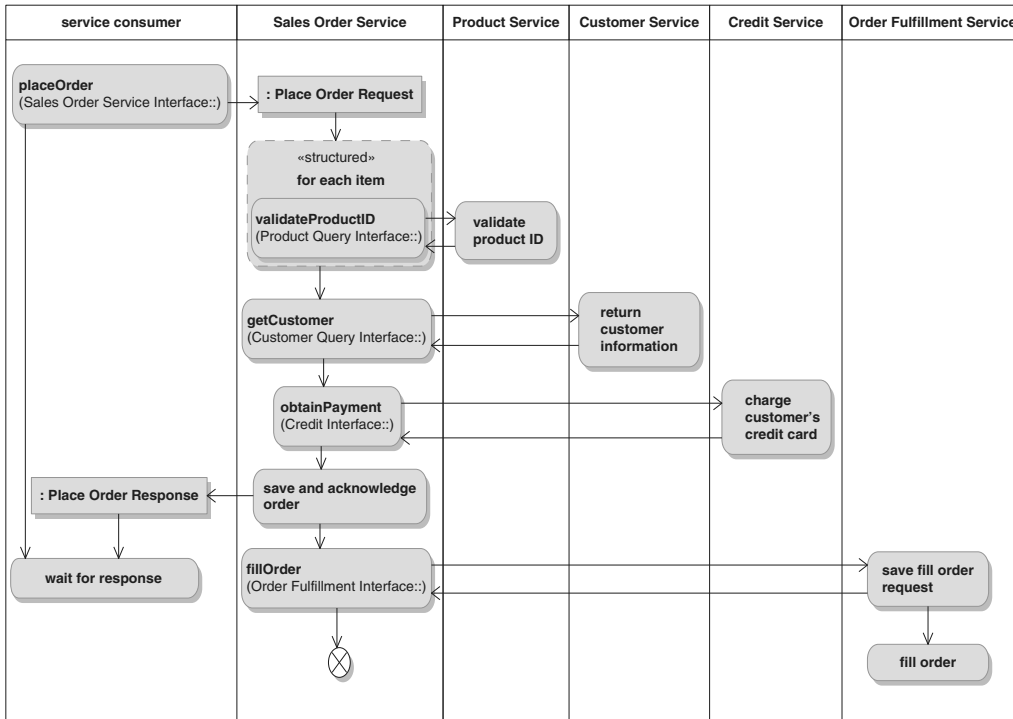


Figure 5-7: *Triggered Behavior for placeOrder()*

Order Shipped

When Order Fulfillment ships the items, it sends a copy of the shipment notice to the Sales Order Service. It does this by calling the `orderShipped()` operation of the Sales Order Status Interface (Figure 5-8). This is another dependency: Order Fulfillment depends on this interface. As designed, this is an In-Only operation.

The triggered behavior related to this interaction, the update of order status, is simple, although somewhat obscure from an observability perspective (Figure 5-9). The obscurity arises because the update of the order status cannot be inferred from this interaction alone. It is only the fact that the order status can be retrieved (via other operations), coupled with the fact that this status indicates whether or not the order has shipped, that reveals the fact that the order status exists and has been changed. We have identified an element of observable state.

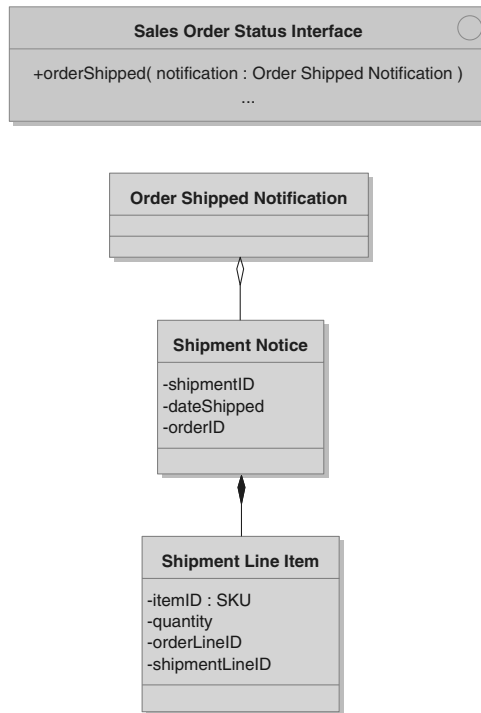


Figure 5-8: *Sales Order Status Interface—orderShipped() Operation*

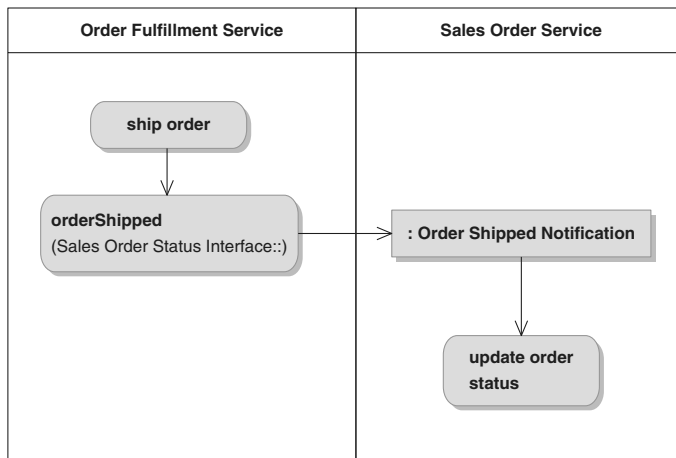


Figure 5-9: *orderShipped() Triggered Behavior*

Order Delivered

When the Carrier reports that the shipment has been delivered, Order Fulfillment forwards the delivery notice to the Sales Order Service by calling the `orderDelivered()` operation of the Sales Order Status Interface (Figure 5-10). This is another dependency. Note that this is an In-Only operation and does not return a response.

Figure 5-11 shows the triggered behavior resulting from the invocation of the `orderDelivered()` operation. Once again, the existence of the order status update activity is inferred from information visible through other interface operations.

Observable State Information

Some component operations can reveal that information has been retained within a component. Consider the `getOrder()` operation of the Sales Order Service Interface shown in Figure 5-12. It returns information about the sales order and its line items. In order for the

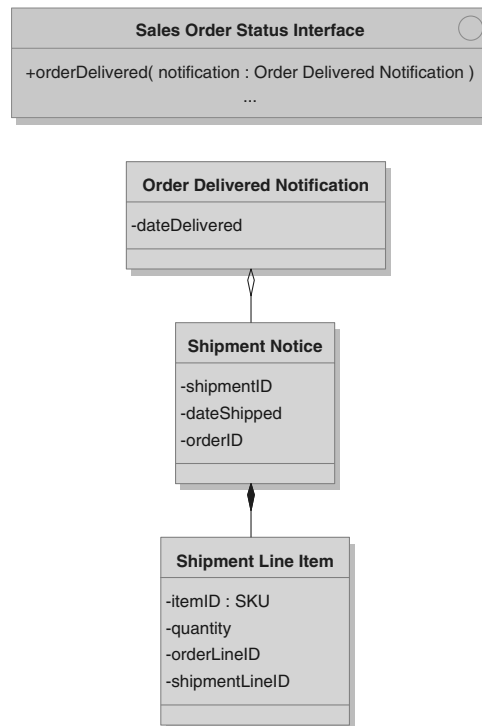


Figure 5-10: *Sales Order Status Interface—`orderDelivered()` Operation*

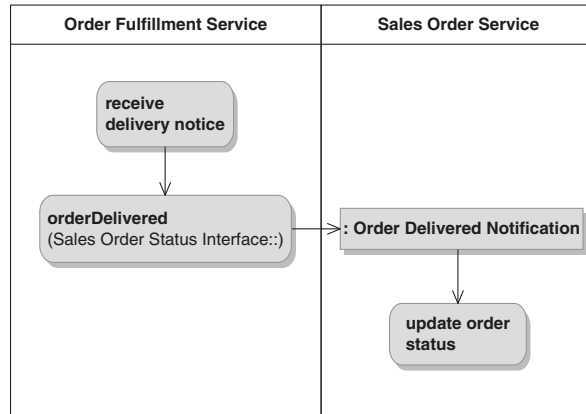


Figure 5-11: `orderDelivered()` *Triggered Behavior*

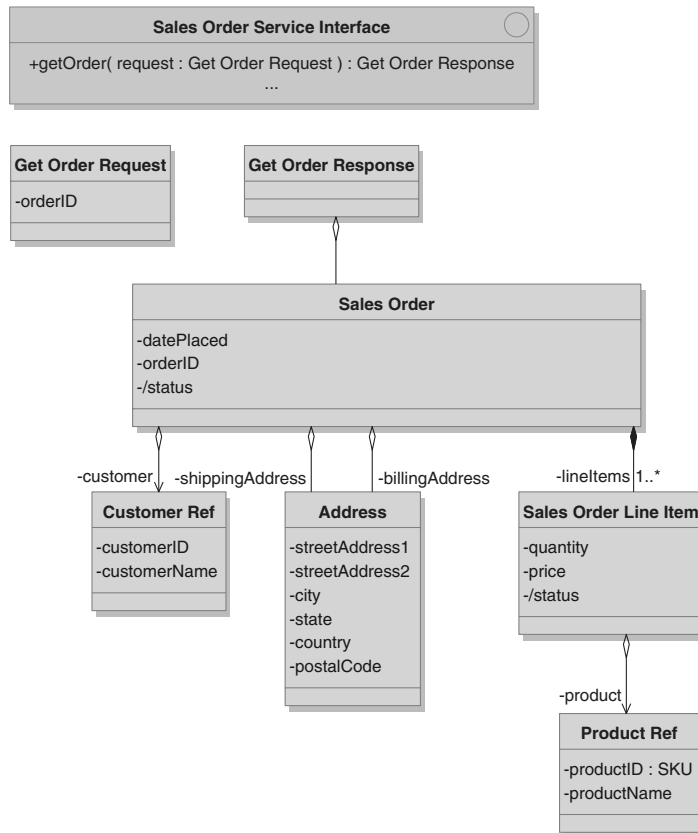


Figure 5-12: *Sales Order Service Interface—`getOrder()` Operation*

component to be able to return this information, it must retain it as part of its state. This makes that portion of the state observable.

There are two types of information typically observable through such interfaces: operational information and milestone-level status.

Operational Information

Figure 5-13 shows, at a conceptual level, the operational information involved in the ordering and shipping of goods. It also indicates which components are responsible for managing individual information elements. The Sales Order Service manages operational information related to the sales order, including the Sales Order, the Sales Order Line Items, the billing information, and the shipping and billing addresses. The service is stateful since it retains this information. The fact that the service makes this information (and changes to it) visible at its interfaces makes this state information observable.

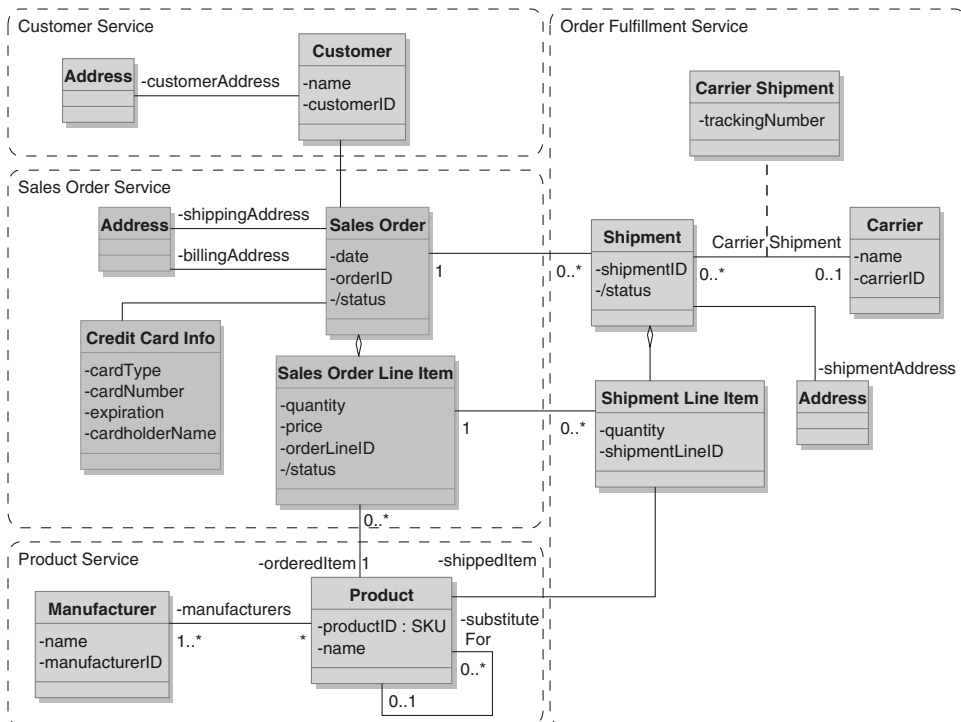


Figure 5-13: Operational Information in the Order-to-Delivery Process

When a component retains stateful information, component users need to know the relationship between the interface operations and this stateful information in order to use the operations correctly. Users need to know which operations modify and reveal the state, and the details about which portions of the state are modified or revealed.

Milestone Status

There is another kind of information often visible through service interfaces: milestone-level status. This is usually an abstracted summary of the overall solution state, which includes state information originating outside the component.

For example, the status attribute of the sales order takes on one of a number of values depending upon the overall status of the order (Figure 5-14). Much of the state information being summarized resides outside the scope of the Sales Order Service. Thus, there must be interfaces (and systems implementing or invoking those interfaces) that provide the detailed state information needed to update this summary

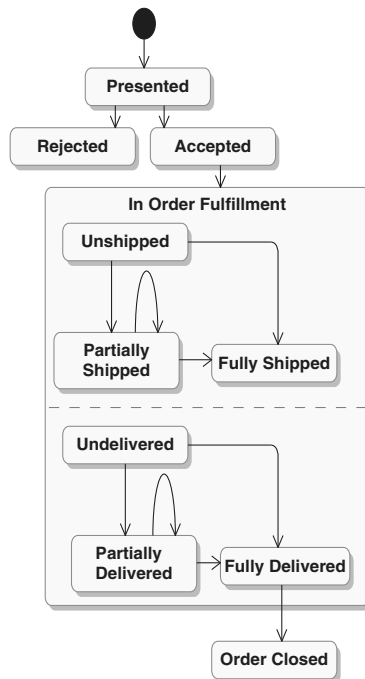


Figure 5-14: *Order Milestone Status*

state information. In this case, these are the `orderShipped()` and `orderDelivered()` operations of the Sales Order Status Interface that are invoked by the Order Fulfillment Service.

Observable State and Cached Information

The Sales Order Service makes use of some state information that it does not directly manage (own): customer information, product information, and information about the related shipments. Other services (components) are the systems of record for this information, but at least some of this information is cached in the Sales Order Service. This situation can raise some interesting design challenges, challenges that when resolved, can impact the observable behavior of the Sales Order Service.

A core design challenge is deciding how to maintain consistency between the cached information as viewed through the Sales Order Service and the same information viewed through its actual system of record. If it is possible (and it almost always is) for inconsistencies to arise, then the component's observable behaviors must indicate the scenarios under which this can arise. Users need to be aware that such inconsistencies are possible and the circumstances under which they can arise.

Let's take a look at how such a situation can arise in the Sales Order Service. If the other systems of record have separate data stores (e.g., databases), then it must be the case that the Sales Order Service retains copies of at least some of the information in its physical data store (Figure 5-15). Since this information is a copy, inconsistencies will arise if the system of record is updated but the copy is not. The more information that is copied, the more likely it is that a discrepancy will arise and be observed.

Maintaining the accuracy of cached information requires interactions with the information's system of record. One common approach is for the system of record to provide facilities to inform interested parties of changes to its information. The system of record provides an interface for interested parties to subscribe to such notifications, and a second interface to actually notify the parties of changes. This approach is often taken when it is likely that more than one party will be interested in the changes. Note that the uses of these interfaces constitute additional observable dependencies.

In the present design, the Sales Order Service has two relationships of this type, one for product information and the other for

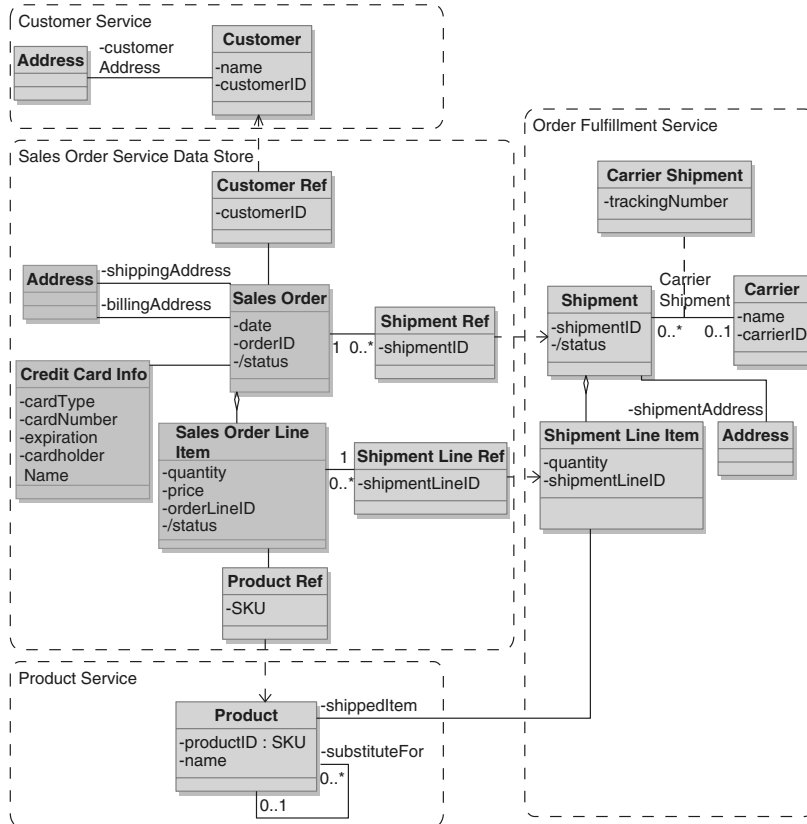


Figure 5-15: *Sales Order Service Data Store*

customer information. Figure 5-16 shows the interfaces provided by the Product Service for this purpose. The Sales Order Service is a user of these interfaces.

Two interesting questions arise with respect to the subscription interface. The first of these relates to the granularity of the subscription: Does the interested party subscribe to changes to particular products, or to all products? The second relates to the timing of the subscription: When does subscription occur? When the component is deployed? When it is started? Does it occur at some other time?

In practice, subscriptions are often realized without implementing subscription interfaces at all. Instead, the design uses a messaging service (e.g., JMS) for the delivery of notifications. The granularity issue is addressed through the choice of the number of destinations (topics or queues) in the design and the determination of which

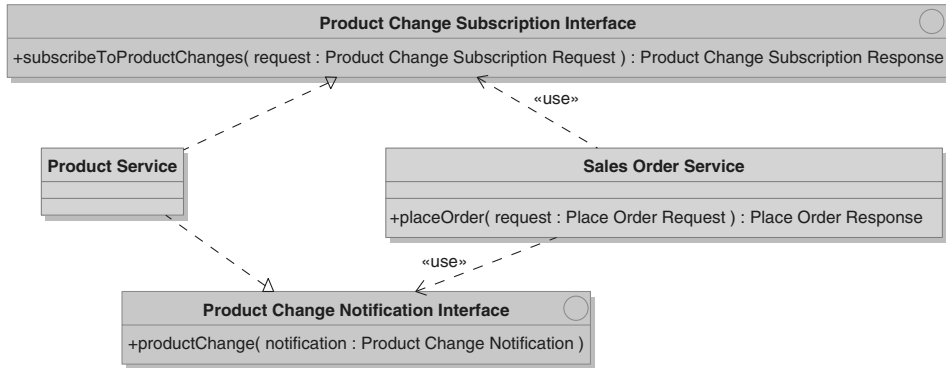


Figure 5-16: *Product Change Notification Context*

notifications will be sent to which destinations. Subscriptions are implemented through deployment-time configuration of components as listeners to specific destinations.

The `productChange()` operation raises another interesting situation from the Sales Order Service's perspective: its invocation triggers activity in the service that is not related to an operation being provided by the Sales Order Service itself (Figure 5-17). It is the arrival of the

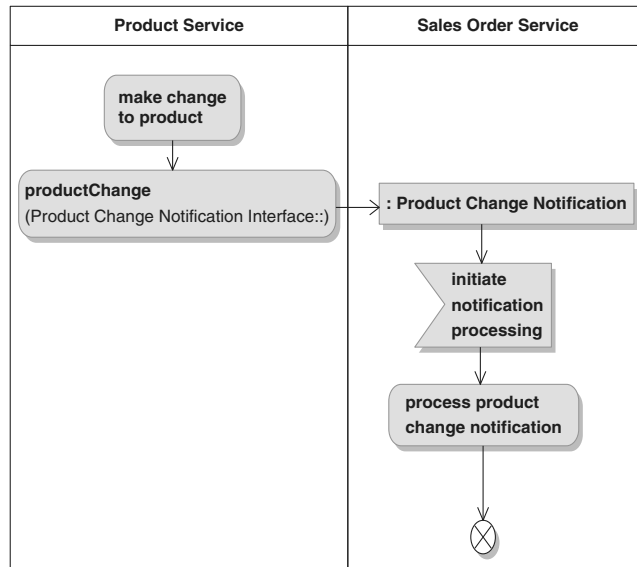


Figure 5-17: *Product Change Notification Process*

notification that triggers the service's activity. This is commonly referred to as an event-driven interaction. As shown, the process depicted in the diagram does not indicate what action should be taken as a result of this notification. However, if the final resolution requires either a change to the observable state of the service (e.g., replacing the item with another item) or an interaction with an external component (such as sending an e-mail notification), these actions constitute changes to the observable behavior that must be documented.

Avoiding Caches: Nested Retrieval

An approach to minimizing inconsistencies is to minimize the amount of cached information in a component. For example, instead of caching a lot of data, the component might only cache an identifier. Then, when the service needs more information about the identified entity, it retrieves it dynamically from the system of record for that entity.

The `placeOrder()` operation described earlier contains two interactions of this type. First, it interacts with the Product Service to validate the `productID` in the order request. Second, it interacts with the Customer Service for two reasons: to validate the `customerID`, and to retrieve the `customerName`, which is a required field for the Fill Order Request (Figure 5-6) data structure used in the `fillOrder()` invocation. Note that only the identifiers for these two entities are retained as part of the Sales Order Service's state.

Characterizing Observable Dependencies and Behaviors

Let's now summarize the information needed to characterize the observable dependencies and behavior of a component.

Context

Context places the component in question into the larger environment in which it must exist and with which it must interact. It defines the dependencies that the component has upon other components. For services, the consumer of the service is often shown only as an abstraction since there may be many service consumers. For components that are not services, the component may require specific

interfaces on the consumer and is designed to work only with that consuming component. In such cases the type of the consuming component is explicitly shown.

Dependencies can be readily shown with an abstracted architecture pattern (Figure 5-18). The difference between this and a full architecture pattern is that the actual communications channels have been replaced with the more abstract <<use>> relationship. As the design is refined, these can be replaced with the more concrete communications channels.

This particular diagram indicates another area requiring refinement: The actual mechanisms for subscribing to the Product Change Notification and Customer Change Notification have not been defined, nor is it clear which participant will employ this mechanism to establish the subscriptions. In reality, the implementation of this activity may require manual configuration done at deployment time. The completed component dependency and behavioral description must indicate how this will happen.

Usage Scenarios

The architecture pattern does not indicate how the component (in this case the Sales Order Service) functionally participates in the business processes that comprise the overall solution. For this you need to

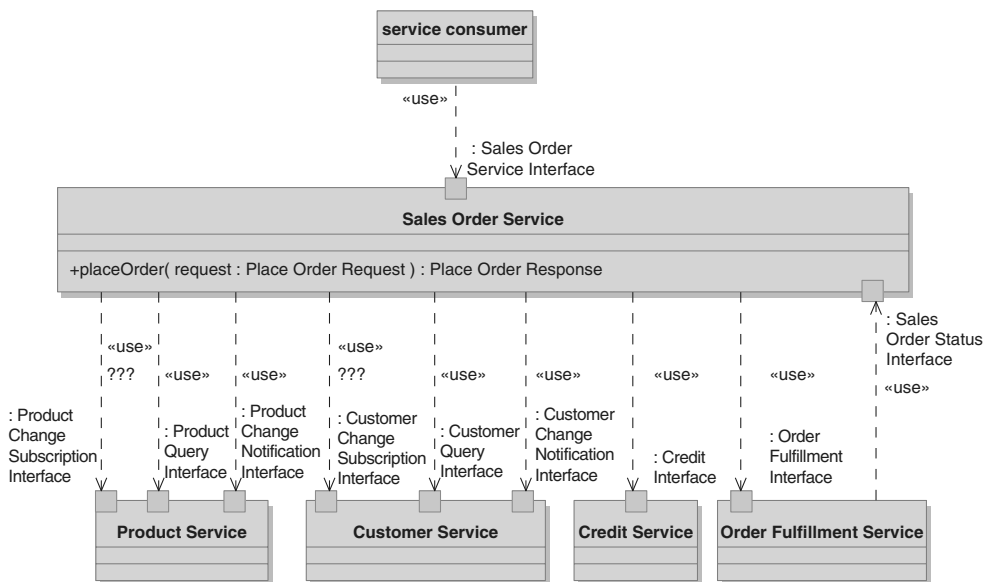


Figure 5-18: *Sales Order Service Context Showing Dependencies*

understand the scenarios that involve the component and show its involvement in the solution's operation. Process-pattern mappings similar to that of Figure 5-1 are well suited for this purpose. For each triggering event of the component, there should be at least one example of a scenario in which the component is expected to participate. If reusability is an issue and the usages are different, there should be a scenario illustrating each type of usage.

To fully characterize the component, more is required than simply the scenario. You need to know how often each scenario occurs, the expected execution time of the scenario, and the required availability of the scenario. It is from this information that the corresponding throughput, response time, and availability characteristics of the component will be derived.

Particularly in the case of services, it is not unusual for some of the scenarios to be speculative, representing potential future usages. Nevertheless, these scenarios need to be documented along with working assumptions about their associated performance and availability characteristics.

From a practical perspective, usage scenarios may only show the fragment of the larger business process in which the component actually participates. However, if the process requires multiple interactions with the component, it is important that the usage scenario span these multiple interactions. If significantly different sequences must be supported, each sequence must be documented.

Triggered Behaviors

The behavior of a component is a description of the sequences of interactions it can have with the components upon which it depends along with the details of those interactions. In most circumstances, this behavior can be readily documented using one or more UML Activity diagrams. The diagrams indicate the behavior's trigger along with the resulting responses, inputs, outputs, and observable state changes.

Figure 5-7 is an example of a triggered behavior. Its trigger is the invocation of the `placeOrder()` process, and its initial input is the Place Order Request. Responses include the calls to `validateProductID()`, `getCustomer()`, and `obtainPayment()`; the return of the Place Order Response; and the invocation of the `fillOrder()` operation. The data structures associated with these operations provide details of the interactions. Prior to sending the Place Order Response, the order information is saved and becomes part of the component's observable state.

This, of course, is just one possible behavior for this trigger. Other scenarios are required to describe the expected behavior when one or more of the dependent components becomes unavailable or returns unexpected results.

To fully characterize a component's behavior, a triggered behavior description is required for each possible trigger. Triggering events may include the invocation of interface operations, the receipt of notifications, the expiration of timers, and component life-cycle events such as start, stop, deployment, and un-deployment. In the Sales Order Service example, it is likely that the subscriptions to product and customer change notifications would actually be made via configuration changes implemented as part of the deployment process. Here the deployment would be the event, and one of the participants in the process is the person doing the deployment.

Observable State

The observable state of a component reflects the information or other types of status (such as physical machine state) that can be altered or viewed by interactions among the component and other components. A model of this state information will help the user of the component understand the component's behavior. The model should clearly distinguish between information for which the component is the system of record and information that is a cached copy of information originating in another component.

Figure 5-15 is an example of an observable state model related to the Sales Order Service. It shows the information for which the Sales Order Service is the system of record and the information that it has cached from other components. It also indicates the relationship between the cached information and the system-of-record information from which it is derived.

Some state information can be a derived summary of information that is distributed across a number of components. The status of the Sales Order is such an example. When this type of information is present, the allowed values that it can assume must be modeled (Figure 5-14), and the triggers and triggered behaviors that result in its update must also be captured (Figures 5-9 and 5-11).

Coordination

The work of a component does not occur in isolation, and the performance of this work needs to be coordinated with that of other compo-

nents in the solution. Consequently, the available coordination approaches are a significant part of the component's observable behavior.

Some coordination patterns are readily captured in the modeling of individual triggered behaviors. For example, in the `placeOrder()` process of Figure 5-7 it is clear that the interaction between the service consumer and the Sales Order Service uses synchronous request-reply coordination.

Other coordination patterns may involve multiple triggering events and therefore multiple triggered behaviors. For example, `placeOrder()` sends a `fillOrder()` request to the Order Fulfillment Service, but the responses from the Order Fulfillment Service are returned asynchronously. These interactions involve the Order Fulfillment Service's invocations of the `orderShipped()` (Figure 5-9) and `orderDelivered()` (Figure 5-11) operations. The overall coordination is only apparent when the usage scenario (Figure 5-1) is considered.

Capturing coordination is important because changing coordination patterns involves changes in both components. In the Sales Order Service example, shipment and delivery notices are delivered to the Sales Order Service by calling operations on its Sales Order Status Interface. This makes the design of the Order Fulfillment Service specific to the Sales Order Service.

There is an alternative approach. Consider a situation in which other components in addition to the Sales Order Service need to know about shipments and deliveries. With the present design, accommodating this requirement would necessitate an Order Fulfillment Service change to individually notify each of the additional components.

Alternatively, the Order Fulfillment Service could provide a subscription interface where any component could register to be informed about shipments and deliveries (Figure 5-19). Thus any number of components could subscribe without requiring any design changes in the Order Fulfillment Service. However, to switch to this design the Sales Order Service has to be modified to utilize the new approach to learning about shipments and notifications.

Constraints

Usage scenarios show allowed sequences of interactions with the component, but they do not illustrate sequences that are not allowed. These need to be documented as well.

Consider the Sales Order Service Interface shown in Figure 5-20. This interface has some obvious constraints upon its usage. You can't get, modify, or cancel an order that hasn't been placed. However, there

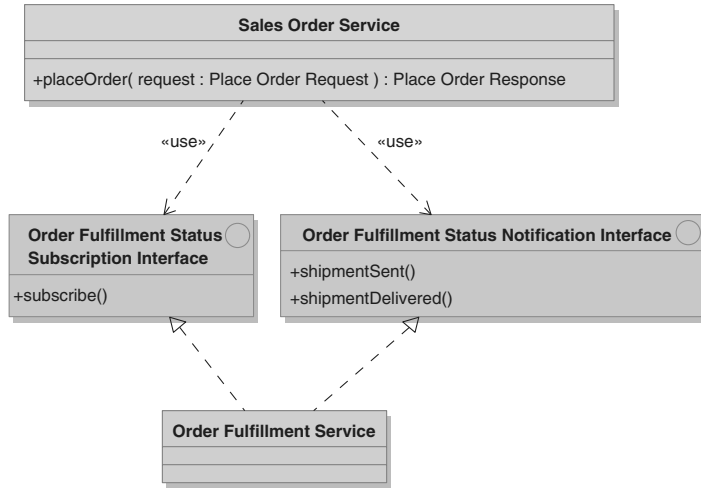


Figure 5-19: *Alternative Design for Shipment and Delivery Notification*

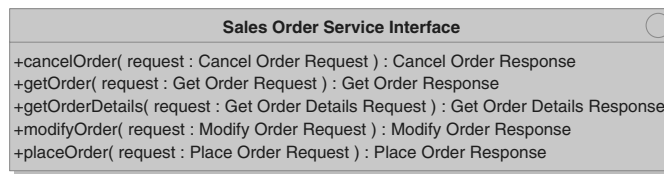


Figure 5-20: *Full Sales Order Service Interface*

may be some less obvious constraints. Depending upon business rules, you may not be able to modify or cancel an order that has already shipped. Users of a component need to understand these constraints.

Nonfunctional Behavior

A component may provide all the functionality required for a usage scenario, but still may not be suitable for nonfunctional reasons. Its throughput capability may be insufficient to support the volume of activity required by the scenario or its response time may be inadequate. The availability of the component may not be sufficient to give the usage scenario its required availability.

Nonfunctional requirements are not arbitrary—they are (or should be) derived from the business requirements. The connection between the business requirements and the individual components is established through the usage scenarios. For example, the business might

require that customers be able to place orders at a peak rate of 100 per second. With reference to Figure 5-1, this means that the Sales Order Service must be able to accept order requests at this rate. If the business requires that orders be acknowledged within three seconds, this means that the Sales Order Service must be able to validate orders and obtain payment within three seconds with orders coming in at a rate of 100 per second. Similar reasoning can be used to determine the rate at which shipment and delivery notices will occur and be processed.

This same type of thinking applies to other types of nonfunctional requirements as well. If the business requires that online ordering capability be available 24/7, then this means that the Sales Order Service must be available 24/7. Availability, outage time restrictions, and security requirements must also be connected back to the business requirements via the usage scenarios.

There is another reason for establishing this connection between business requirements and component requirements: It captures the design assumptions that went into specifying the component. When a new utilization for the service comes along, this makes it easy to determine whether the new usage is consistent with the original design assumptions. If it is not, then it is necessary to open the black box and determine whether the actual design is capable of meeting the new requirements.

For all of these reasons, it is important to document the nonfunctional behavior of the component. It is an observable characteristic of the component.

Some Composites May Not Be Suitable for Black Box Characterization

For the black box approach to work, the component must appear to be a coherent whole. For a monolithic self-contained component, that's not a problem. But for a composite, a component comprising other components (Figure 5-21), some conditions must be satisfied for it to be treated as a black box:

- It must (to all appearances) be managed as a single unit.
- It must have a single organization responsible for its operation.
- Access to the constituent parts must be exclusively through the interface(s) of the black box. Note that this does not preclude exposing a sub-component's interface as one of the composite's interfaces.

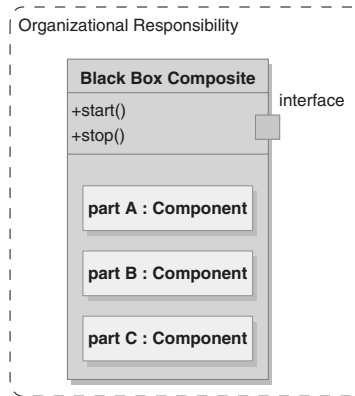


Figure 5-21: *Black Box Perspective*

Key indicators that the composite is not a black box include

- Different organizations are responsible for operating (starting and stopping) different parts of the composite.
- Decisions to start and stop the composite are separate from decisions to start and stop its component parts.
- A component part can be accessed directly via its own interfaces, interfaces that are not declared to be part of the composite's interfaces. This is particularly important when the constituent component is stateful.

An example of a component that should not be treated as a black box is a service (interface) wrapping the functionality of a stateful back-end system that other solution components access directly. In this situation, state changes made via the service interface may be visible through the back-end system's interface, and vice versa. Characterizing the observable behavior of the service will not explain the state changes that are visible through the service interface but did not originate as the result of a service interaction.

Any of these conditions make it important for the user of the composite to be aware of the composite's architectural structure—aware of those components that are independently accessible or managed. Under such circumstances, you can't treat the composite, including its component parts, as a single entity—a black box. Instead, each of the components becomes a black box in its own right. The context and usage scenarios for the "composite" then indicate how these supporting components collaborate with the composite.

Summary

When you want to use a component as part of a solution, you need to understand the behavior of that component so that you can determine whether it is suitable for use in the solution. What you need to know are the aspects of the component's behavior that are observable from outside the component. You treat the component as a black box and focus on its observable behavior.

Understanding observable behavior requires characterizing a number of things, including

- Context: the component's dependencies upon external components and vice versa
- Usage scenarios: characterizations of the business processes in which the component is expected to participate
- Triggered behaviors: structures of activities that explain the relationships among the component's triggers, responses, inputs, observable state, and outputs
- Observable state (information retained from one trigger to the next)
- Coordination: the manner in which the component's activity can be coordinated with that of external components
- Constraints: limitations, particularly on the sequencing of triggers
- Nonfunctional Behavior: performance, availability, and so on

Some composites cannot be safely considered as black boxes. These include components whose constituent parts are operated independently or are accessible by means other than the composite's interfaces.

Naming Standards

Objectives

This chapter provides guidance for an enterprise architect defining the standardized structure of names for web services and other IT elements in the enterprise. It explores the issues and best practices involved in defining and managing enterprise naming standards. The topic is of particular importance because the structuring of the names and the challenges involved in maintaining the consistency in naming WSDL and XML schema namespaces, services, ports, operations, and other artifacts in the web services space have a significant impact on the complexity of managing an SOA environment.

Although the discussion and examples are focused largely on web services (i.e., services defined with a WSDL), the basic naming principles also apply to schemas, JMS destination names, uniform resource identifiers (URIs) and uniform resource locators (URLs). Several of the important names in a WSDL are required to be URIs.

There are many possible approaches to designing names for use in SOA environments. This chapter sets forth one concise set of best-practice concepts for designing names that can be readily tailored to the needs of your enterprise. In explaining the concepts, particular emphasis is placed on explaining the rationale behind the approach so that the refinements that will inevitably be required in practice can be designed with the same principles.

After reading this chapter you should be able to

- Describe the principles guiding name structure design.
- Apply the principles to the ideal design of names for WSDL and schema artifacts.
- Describe the choices available for addressing real-world complications in name structure design.

Using This Chapter

The intent of this chapter is to provide guidance in the formulation of naming standards. It begins with a discussion of the concepts and general principles for structuring names. These principles represent a rationale for structuring names that should, as a rule, always be adhered to.

But principles are not enough: There are circumstances under which the principles alone will not provide a unique solution. As these cases arise in this chapter, guidelines are provided to indicate reasonable ways of addressing the situation and guidance for selecting an appropriate approach.

Next, some complicating realities are discussed. Practical constraints imposed by the implementation technology constrain the structures of names. Organizational issues of various types introduce complications that require guidelines for resolution. The existence of multiple environments (development, test, production) and the need for flexible deployment (for fault tolerance, high availability, site disaster recovery, or simply administrative convenience) add wrinkles of their own. Versioning must also be taken into consideration.

The chapter concludes with some practical guidance for creating SOA naming standards.

Concepts

Abstract Services

Any discussion of service naming standards has to begin by putting a stake in the ground in terms of defining what a service is. Here you run into a difference of abstraction between the working definitions used in

the broader SOA community and those provided specifically by web services (i.e., defined by the WSDL schemas).

Thomas Erl provides the following statement that is generally representative of the broader SOA community perspective: “Each [abstract] service is assigned its own distinct functional context and is comprised of a set of capabilities related to this context. Those capabilities suitable for invocation by external consumer programs are commonly expressed via a published service [interface] contract (much like a traditional API).”¹

The structure of the concepts in this statement is shown in Figure 8-1. Here the term *Interface Contract* is used to reinforce the fact that the concept being represented is an interface. This also emphasizes that an Abstract Service may have multiple interfaces. This concept structure provides a useful framework for exploring the structure of names.

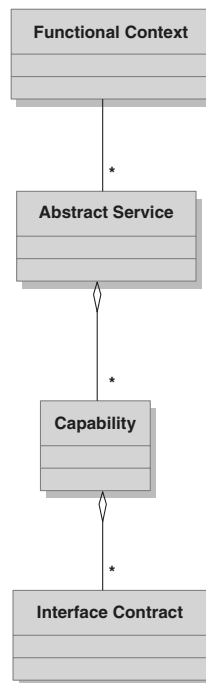


Figure 8-1: *Abstract Service Concepts*

1. Thomas Erl, *SOA: Principles of Service Design*, Upper Saddle River, NJ: Prentice Hall (2008), p. 39.

WSDL Interface Definitions

The Web Services Description Language (WSDL) is designed to define service interfaces. A WSDL file contains two different kinds of definitions: abstract definitions (often referred to as types or meta-data) and concrete definitions (Figure 8-2). The abstract definitions specify the types of portTypes (interfaces), operations, messages, and message parts. The message parts, in turn, reference schema data types (see the sidebar on incorporating schema data types). The concrete specifications define the instances of services, ports (endpoints), and their bindings to protocols. WSDL and schema definitions each occur in a context uniquely identified by their respective namespace URIs.

Two of the names are particularly important: the address of the port (endpoint) and the `soapAction` of the operation binding. Their importance stems from the fact that these are the two primary names used in the communications between the service consumer and provider when SOAP bindings are being used. The address specifies the place to which messages are being sent, and the `soapAction` is the sole indication of the operation that is being invoked. It is from these two pieces of information that the recipient determines the abstract definitions (meta-data) that characterize the message structure and the operation to be performed.

The WSDL part definitions reference data types from a schema definition. It is a best practice for each message part to have its own dedicated schema type associated with it. This allows each data structure to be specific to the operation it supports and avoids unnecessary updates if these data types are shared. The namespace URI for this schema should be the same as the WSDL namespace URI except for their respective endings (schema or WSDL). The use of different endings is primarily for human readability.

Best Practice: Dedicated Data Types for Message Parts

Use a different dedicated data type for each message part, and define these data types in a schema dedicated to the WSDL. The namespace for the schema and WSDL should be the same except for their respective endings (schema or WSDL).

The dedicated data types for message parts may well contain elements having data types that belong to a common data model. Those

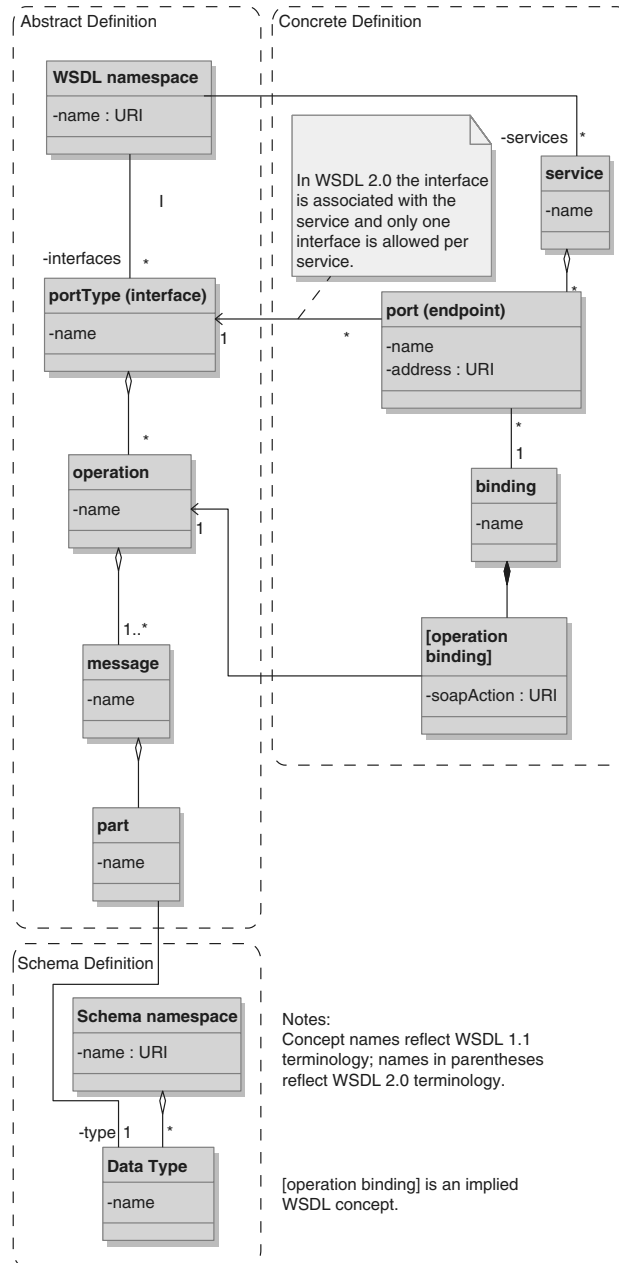


Figure 8-2: *Simplified WSDL Concepts*

data types, which are designed to be shared, should be defined in their own schema, one that has a definition that will be in its own XSD file. The namespace for the common data model XSD should reflect the intended scope of utilization of the contained data types (more on this later).

Best Practice: Incorporate Schema Data Types, Not Elements

Definitions that occur in an XML Schema Definition (XSD) are often incorporated into other schema and WSDL definitions. Both WSDL and XSD standards provide two options for doing this. One is to incorporate a data type, in which case the WSDL or XSD incorporating the type defines a local element of that type. The other is to incorporate an element definition and use it as-is—including its name. The best practice is to incorporate the data type.

The rationale behind this best practice is that the name of an element reflects the role that the element plays in the context in which it is being used. The author of the shared schema has no way of knowing how those definitions are going to be employed and therefore may not be in a position to choose appropriate element names. Furthermore, new usages may require new names.

Consider the concept of an address, a likely candidate for a shared schema definition. If the shared schema defines an Address data type, then it is easy for one consuming data structure to define a homeAddress element and another to define a workAddress element.

This approach is open ended: You do not need to know all of the usages ahead of time to create the correct element names. You can later create other data structures that define a shippingAddress or billingAddress, or any other address role you can imagine. The flexibility occurs because these roles are defined in the new data structures—not the shared schema.

The alternative, creating the elements in the shared schema, not only requires a change to the shared schema each time a new role is added (which requires a new element with that role name), but these shared schema changes impact all of the other consumers of the shared schema. For this reason, the best practice is to incorporate data types, not elements.

Relating Abstract and WSDL-Defined Services

The correspondence between Erl's abstract concepts and the WSDL definitions is shown in Figure 8-3. The functional context corresponds to a functional area that may contain many WSDL-defined service interfaces. The subdivisions of the functional context, the abstract

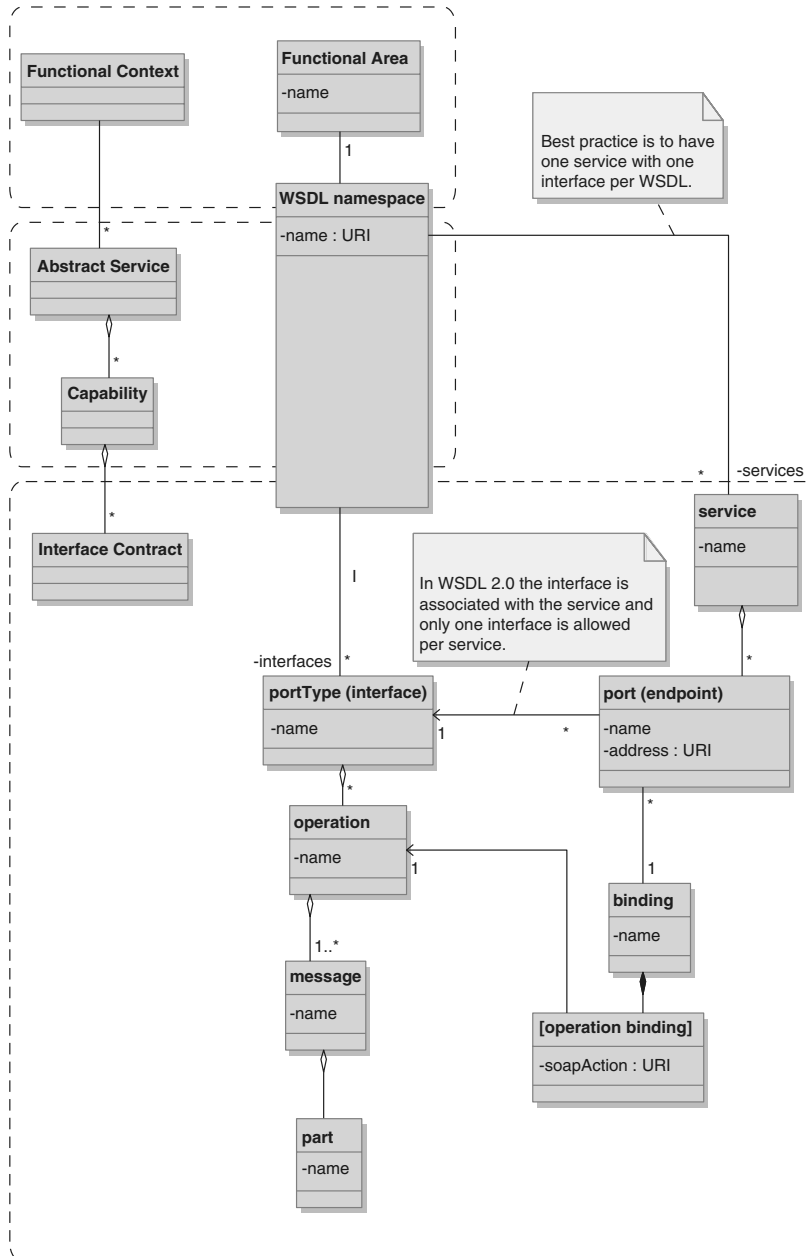


Figure 8-3: Correspondence between Abstract Services and WSDL Definitions

service and capability, are largely implicit in the structure of the WSDL namespace (this will be discussed in detail later). The interface contract encompasses the interface definition but, as was discussed in Chapter 6, there is a lot more to the contract than just the interface definition. The interface contract may encompass multiple WSDLs, particularly if the best practice recommendation (Chapter 7) of only having one interface per WSDL is followed. *In the ensuing discussion, unless explicitly stated otherwise, the term service is a reference to the abstract service concept, not the WSDL definition.*

Why Are Names Important?

When you use a service, that service and its operations need to be readily identifiable and distinguishable from other services and operations. This is accomplished by giving a unique name to each abstract service and, within the service, to each port type (interface) and operation of that service. Unique names are also required for the messages, data structures, and other artifacts used in the definition of the service interface. When you use JMS destinations (queues and topics), it is also important to distinguish them from other destinations. This gives you the flexibility to move these destinations to other EMS servers.

There are two significant challenges that you will encounter when creating names:

1. Ensuring that each name is reasonably descriptive
2. Ensuring that each name is unique

Naming standards establish how names are structured and organized and how (by whom) uniqueness is guaranteed, and thus determine how both challenges are addressed.

Names Are Difficult to Change

Unfortunately, once established, names tend to become deeply embedded in service providers, service consumers, and the components that mediate interactions between the two. Thus changing established names is expensive—often prohibitively so. You are going to be stuck with the names you choose for a long time, so it is worth an investment of time in considering the standards that govern their structure.

Name Structures Define Search Strategies

In SOA, the structure of a name not only identifies the item being named, but often indicates how to find the item. When people are looking for an artifact, they often begin with only a vague notion of what they are looking for. While indexes and keyword associations can be of use, the structure of the name implicitly defines a search strategy for locating items. Consider, for example, the Nouveau Health Care Payment Manager's Claim Payment Interface. Its WSDL might have the URL

```
http://insurance.nouveau.com/finance/paymentManager/  
claimPayment/wsdl/claimPayment.wsdl
```

This name not only uniquely identifies the WSDL file, it tells us where to find it: Go to the machine designated by `insurance.nouveau.com` (at the default port 80), look in the `finance/paymentManager/claimPayment/wsdl` directory, and obtain the file `ClaimPayment.wsdl`.

Name Structures Define Routing Strategies

Although it is not formally a part of the web services paradigm, ports (endpoints) often make use of names in directing service requests to the actual service provider. The component (and often the machine) that accepts service requests is often different than the component actually servicing those requests. Appropriate naming can aid the receiving component in routing requests to the provider.

For example, the port (endpoint) address (the address to which service requests are directed) is designated by a URI, which is a form of name. Consider the address of the claim payment interface for Nouveau Health Care, given as

```
https://insurance.nouveau.com:443/finance/paymentManager/  
claimPayment
```

Also consider the address of the claim submission interface, given as

```
https://insurance.nouveau.com:443/claims/claimRouter/  
claimSubmission
```

The actual component to which both kinds of requests are directed is designated by the first part of the address URI, namely the domain name and port (not to be confused with the WSDL concept of port):

```
https://insurance.nouveau.com:443
```


This indicates that requests are received on port 443 of the machine designated by `insurance.nouveau.com`. From there, if the requests are to be serviced by another component, they must be routed to the actual service provider. The presence of the URI path structure in the address below the Internet domain name and port number makes it possible to route requests arriving at port 443 based on this path. Alternately, this information can be ignored and all requests can be routed to a common destination that provides support for all web services sharing the domain name and port.

In contrast, both addresses could have been given as

```
https://insurance.nouveau.com:443
```

In this case, since the path is absent, there is no information that can be used to route requests.

Using the structure of the address is, of course, not the only way to route. Other information such as the `soapAction`, the structure of the WSDL namespace URI, and the port/operation naming structure underneath it can also be used for routing. Regardless of which names are being used, the structure of these names and the consistency with which that structure is used can greatly simplify the routing of service requests.

What Needs a Name?

In an SOA environment there are many things that require names. These include

- WSDL-related names:
 - The WSDL itself (the `name` attribute in the top-level `<definitions>` tag).
 - The WSDL filename.
 - The `targetNamespace` defined by the WSDL (also in the top-level `<definitions>` tag). This name is a URI, and is the logical prefix for the name of each entity defined in the WSDL.
 - The entities defined by the WSDL: services, port types (interface), operations, messages, bindings, and ports (endpoints).
 - The `soapAction`.
 - The `schemaLocation` of each imported XSD file.
 - The address of each service port (endpoint).

- Schema-related names:
 - The XSD filename (if the schema is not embedded in a WSDL).
 - The `targetNamespace` defined by the schema (either embedded in the WSDL or a stand-alone XSD). This name is a URI and is the logical prefix for the name of each entity defined in the schema.
 - The entities defined by the schema: types, elements, and attributes.
 - The `schemaLocation` of each imported XSD file.
- JMS-related names:
 - Static destination (topic or queue) names
 - EMS server names
- SCA-related names
 - Composite names
 - Component type names
 - Service names

Some of these items are actual artifacts, such as files. Others are simply logical names for definitions (port types, messages, elements, etc.) or groups of definitions (namespace URIs). All of them will have to be dealt with by the producers and consumers of services. These names, and the standards for defining them, are the focus of this chapter.

Structured Name Design Principles

Use a General-to-Specific Structure

Structured names (think `www.nouveau.com`) are trees. The root of the tree represents the entire tree. Each node below represents a subtree rooted at that node. The root of the Internet (the parent of `com`) is not explicitly named, but the next tier down in the Internet naming system (referred to as top-level domains) consists of nodes such as `.com`, `.net`, `.org`, and `.edu`. Within each of those domains there are subdomains whose names are meaningful only in the context of the parent domain. Thus for uniqueness you refer to `nouveau.com` rather than just `nouveau` (which has different meanings in `.com`, `.net`, `.org`, etc.).

Because of the inherent tree structure, it is important that the logic of the structure you define be organized around a true hierarchy. The outline of such a hierarchy for services derives from the description of a service presented in the concepts discussion earlier. From the abstract concept discussion we have

```
<functionalContext><abstractService><abstractInterface>
```

This is a general-to-specific structure. A functional context may contain more than one service, and each service can contain more than one interface. Conversely, services are specific to a particular functional context, and each interface is specific to a particular service.

Ideally, the actual hierarchy is represented by a uniform sequence of fields in a left-to-right, most-general-to-most-specific fashion. This is the pattern you see in the naming of Java classes:

```
com.nouveau.insurance.finance.paymentManager.claimPayment.  
payClaim
```

Here `com.nouveau.insurance.finance` represents the functional context. The abstract service is the `paymentManager`. The abstract interface is represented by `claimPayment.payClaim`, the interface and operation used for paying a claim. Note that there is hierarchical structure both within the functional context and within the interface.

Unfortunately, many names in the SOA world are required to be URIs or URLs. These include namespace names and WSDL/SOAP `address` and `soapAction` fields. The structuring scheme for URIs does not strictly adhere to a uniform principle. Each URI or URL actually contains two name hierarchies, each with different construction rules. Consider the earlier example:

```
http://insurance.nouveau.com/finance/paymentManager/  
claimPayment/wsdl/claimPayment.wsdl
```

There are three parts to this string involving two different name structures:

1. `http://`, the first part, represents a protocol to be used to access the artifact. From a name structure perspective, we can ignore this.
2. `insurance.nouveau.com`, the second part, is an *Internet domain name* that designates a machine on the Internet. In the Internet domain name, hierarchy is read right to left.

3. `/finance/paymentManager/claimPayment/wsdl/claimPayment.wsdl`, the third part, is called the path. It represents a name hierarchy local to the machine designated by the Internet domain name. The path name hierarchy is read left to right.

What makes the use of two different name hierarchies and their differences in right-to-left and left-to-right reading unambiguous is the use of two different field separators. The Internet domain namespace fields are separated by a period (`.`), while the local namespace fields are separated by a forward slash (`/`). The occurrence of the first slash to the right of the Internet domain name is the clue that you have switched naming hierarchies.

If you wanted to make this namespace uniform, reading entirely left to right, you would have to switch the order of the Internet namespace elements:

```
com.nouveau.insurance.finance.paymentManager.claimPayment.wsdl.  
claimPayment.wsdl
```

This is, of course, the kind of structure used for naming Java classes.

Best Practice: General-to-Specific Structure

When the structure of names is not already constrained, establish a uniform left-to-right, general-to-specific hierarchical structure for names following the pattern

```
<functionalContext><abstractService><abstractInterface>
```

When the structure of names is partially constrained (e.g., URLs and URIs), apply this principle to the lower-level structure of the names (i.e., the path in the URL or URI).

Employ Hierarchical Naming Authorities

Names, to be useful, must be unique. Uniqueness requires that some authority keep track of which names have been assigned so that newly issued names can be assured to be different. Unfortunately, having a single authority to keep track of all names is impractical, particularly for names with global scope.

The systems that have evolved (in numerous places) to address this issue involve the use of hierarchical authorities. This strategy involves two elements. The first is a standardized syntactic structure for defining names as sequences of fields. The Internet domain name structure (e.g., `insurance.nouveau.com`) is one example.

The other element is a policy that establishes authorities for maintaining the uniqueness of individual fields at different levels of the hierarchy. For example, in the Internet domain name structure, the authority for the top-level domain names (.com, .net, .org, etc.) is ICANN (the Internet Corporation for Assigned Names and Numbers). For each of the assigned top-level domain names, ICANN also identifies an authority for maintaining the uniqueness of the next-level names. For the generic top-level domains (.com, .net, .org, etc.) ICANN itself is the naming authority. For sponsored top-level domains (.edu, .gov, .mil) this authority has been delegated to other entities—one for each sponsored domain. For example, the .edu domain is administered by EDUCAUSE, the .gov domain by the General Services Administration of the US Government, and the .mil domain by the US Department of Defense Network Information Center.

Each naming authority does two things:

- It directly manages the field level for which it is responsible.
- It establishes the policy for managing the lower-level fields.

It is in this latter point that the hierarchical authority gains its flexibility: A naming authority can delegate the responsibility for managing subordinate namespaces. For our fictional domain, ICANN would not only establish `nouveau.com` as a domain name, it would delegate the authority for managing the lower levels of the hierarchy to Nouveau Health Care Inc. Nouveau Health Care directly manages the next level of names (such as `www.nouveau.com` and `insurance.nouveau.com`) and (if desired) delegates the responsibility of managing the next level of namespace.

This idea can, and should, be extended within the enterprise. Rather than having a centralized authority for managing all names, the central authority establishes a policy for the structure of names, directly manages the top field(s) of this namespace, and delegates the authority for managing the lower-level fields.

Best Practice: Hierarchical Naming Authorities

Do not fully centralize the management of names in the enterprise. Instead, set up a central authority to (a) establish a generic structure for names, (b) assign the values for the top-level branches of the namespace, and (c) designate appropriate authorities to manage the lower-level structure of individual branches.

Base Naming on Stable Concepts

Because of the difficulty in changing both the structure of names and the individual names that have been chosen, it is prudent to base the namespace structure and individual names on concepts that remain stable over time, with the exception of enterprise identifiers.

Domain Concepts

The concepts that are most likely to remain stable over time are those that generically occur in any discussion of the business. In a discussion of the shipping industry, you would tend to use general terms like package, shipment, track, rate, shipper, recipient, and location. In banking, you would use general terms like customer, account, deposit, withdrawal, payment, and transaction.

What you want to avoid are terms that are likely to change over time, particularly those with branded terminology. Thus you would want to use the generic term “claim” instead of “expressClaim” or “account” instead of “premierAccount.” In doing so, you hedge your bets against mergers, acquisitions, and changes in marketing.

Business Processes

The same rule applies when naming operations associated with business processes. You want to stick with generic names that will remain stable over time. Thus the operation for tracking a claim is called simply “track,” while the marketing-branded process may be called “Nouveau Track.” An idealized (and somewhat simplified) operation name for tracking a Nouveau Health Care claim might be something like

```
com.nouveau.insurance.claims.claimTracker.claimTrack.trackClaim
```

Here the `claimTracker` service presents a `claimTrack` interface with a `trackClaim` operation.

The Exception to the Rule: Enterprise Identifiers

Names, in the context of the world-wide web, must be unique. To guarantee the global uniqueness of names issued by your enterprise, you need to qualify the name with something that identifies the enterprise issuing the name. The most common approach is to base this on the Internet domain name that has been assigned to your enterprise.

This, of course, violates the notion of naming being based on stable concepts—your enterprise could change its name at any time. Mergers and acquisitions are constantly changing this landscape. Despite this potential for change, the use of the enterprise name as a name prefix is necessary. In the examples we have been using so far, ideally structured names in Nouveau Health Care would be prefixed by `com.nouveau`. Note that the order of the fields in this idealized name structure is reversed from the Internet domain names.

The use of the name prefix not only guarantees the global uniqueness of names, but also serves as a hedge against mergers and acquisitions. Without introducing the enterprise-specific prefix, the IT consolidation after a merger or acquisition is liable to create name conflicts. It is likely that two health care insurance providers would both have names like `claimProcessing.claimSubmission.submitClaim`. Adding the prefix averts a conflict with these names as used by the different companies.

Best Practice: Base Naming on Stable Concepts

Base the namespace structure and individual names on concepts that remain stable over time, with the exception of enterprise identifiers.

Avoid Acronyms and Abbreviations

It is good practice to make the names in individual fields of the naming structure as readable and obvious as possible. This argues against using acronyms unless they are widely used across an entire industry and are universally understood. Some common functional abbreviations that might be considered acceptable (at least in English) include

- HR—Human Resources
- MKTG—Marketing
- MFG—Manufacturing
- FIN—Finance

Best Practice: Avoid Acronyms

Avoid using acronyms and abbreviations unless they are universally understood across an industry.

Distinguish Types from Instances

It is not unusual in a WSDL or schema to define a type and then use that type to define an element. For example, you might define a data-type to represent an account balance and then create an instance of that type (an element) as a field in an account to show the current balance. Even if the structure of the defining language allows the same name to be used for both the type and the field name, doing so can be very confusing to a human reader.

Consequently it is a good idea to adopt a standard way of distinguishing type names from instance names. Some examples of such conventions include

- Using leading capitals to indicate types and leading lowercase to indicate instances. Thus the type for the account balance would have the name `AccountBalance`, and the instance field in the account (the element name) would have the name `accountBalance`.
- Appending a distinguishing term to the end of the type name. Thus the type for the account balance would have the name `AccountBalanceType`, and the instance field in the account would have the name `accountBalance`.

It is not unusual to combine the two strategies, as was done in the second example. Note that it is likely that you already have similar standards in place for writing code. If so, you should examine those standards to determine whether they are applicable here as well. If those standards are applicable, you should adopt the existing standard rather than inventing a new one—this will avoid confusion when people are developing code that references WSDL and schema definitions.

Best Practice: Distinguish Types from Instances

Adopt a standard way of distinguishing type names from instance names.

Plan for Multi-Word Fields

It is often useful to be able to use more than one word when constructing a name. This can be accommodated by adopting a convention for capitalizing names that ensures that the resulting multi-word names

are readily readable. Two common conventions that satisfy this criterion are

- Non-leading capitals: Each word in a field other than the first begins with a capital letter, such as `ClaimService` (for a type) or `claimService` (for an instance). The leading capital should be determined by whether the field references a type or an instance (see previous section).
- All letters are capitalized, and an underscore (`_`) or other separator is used between words, for example, `CLAIM_SERVICE`. This convention pretty much requires that an additional term be used to distinguish types from instances (see previous section).

One convention to be avoided is the use of all capitals with no special characters allowed (e.g., no underscores). Such a convention leads to names like `CLAIMSERVICE`, which is not only hard to read but also may lead to ambiguities.

Best Practice: Plan for Multi-Word Fields

Adopt a convention for capitalizing names that ensures that the resulting multi-word names are readily readable.

Use a Distinct WSDL Namespace URI for Each Interface

To avoid naming conflicts among the messages, ports, and operations of different interfaces, each interface should have its own namespace URI. Equally important, using distinct namespace URIs makes it possible to independently version the interfaces. This will facilitate the graceful evolution of interfaces in the environment.

Best Practice: Use a Unique WSDL Namespace URI for Each Interface

Define each interface in a separate WSDL, and use a unique namespace URI for each WSDL.

Incorporate Interface Major Version Numbers

In keeping with the versioning best practices outlined in Chapter 7, wherever the interface name is incorporated into certain naming structures, the interface's <major> version number should be appended to the interface name in those structures. This affects the following items:

- WSDL namespace URLs
- Schema namespace URLs when the schema is dedicated to a specific WSDL (see the section, "Schema Types Specific to an Interface and an Operation" later in this chapter)
- WSDL filenames (also include the interface's <minor> version number, as per versioning best practices)
- Schema filenames when the schema is dedicated to the specific WSDL (also include the interface's <minor> version number in the filename, as per versioning best practices)
- SOAP addresses (endpoints)
- SOAP action names

For simplicity, the version numbers are omitted from the examples in the following sections.

Applying Naming Principles

Idealized Name Structures

Given the structured name design principles laid out above, an idealized form of structure for naming service operations might have the following structure:

```
<functionalContext><abstractService><interfaceName>  
  <operationName>2
```

2. In a WSDL the scope of an operation name is the portType (WSDL 1.1) or interface (WSDL 2.0).

For the data types involved in defining the service, you might use one or more of the following structures depending upon the intended scope of utilization for the data type:

```
<functionalContext><abstractService><interfaceName>
  <operationName><datatypeName>
<functionalContext><abstractService><interfaceName>
  <datatypeName>
<functionalContext><abstractService><conceptPackage>
  <datatypeName>
<functionalContext><conceptPackage><datatypeName>
```

The `<conceptPackage>` is explained later in the section on schema shared data types. Similar idealized structures would apply for the messages, elements, and other artifacts involved in defining the service.

Such idealized structures are rarely seen in their entirety as single strings, with the possible exception of fully qualified Java class names. In most SOA applications, the name structure is fragmented in two different ways:

1. The leading part of the complete name structure appears as the namespace of the WSDL or schema, while the remainder of the structure appears as the name of the artifact being defined within the namespace.
2. Within the namespace URI of the WSDL or schema, part of the name appears as an Internet domain name while the rest appears as the path.

The following sections explore the practical use of these idealized name structures and show how they can be used to guide and standardize the definitions of names. These will be illustrated using the Nouveau Health Care example. Each topic area will first present an idealized name structure and then discuss its practical implementation.

Functional Context

A fact of life is that enterprises frequently acquire or merge with other enterprises. To avoid name conflicts in the ensuing IT consolidation, it is good practice to make the first field in the functional context be an enterprise-specific qualifier, giving

```
<enterpriseID><functionalArea>
```

In practice, the `enterpriseID` is almost always defined using an Internet domain name. For example,

```
nouveau.com
```

Most often the enterprise ID occurs in the context of a URL or URI and has the form

```
http://nouveau.com
```

Many businesses have more than one line of business, with each line of business essentially operating as a separate company (particularly from an IT perspective). When this occurs, it is good practice to include the line of business as part of the `<functionalContext>`. This helps avoid name conflicts among similar functional areas (e.g., sales, human resources, finance) in the different lines of business of the same company. The result is

```
<functionalContext> ::= <enterpriseID><lineOfBusiness>  
    <functionalArea>
```

Consider Nouveau Health Care and the possibility that it might have two very different lines of business. One is an insurance provider, and the other is an online pharmacy. The idealized enterprise identifiers for Nouveau would then be

```
com.nouveau.insurance  
com.nouveau.pharmacy
```

In transforming the idealized form into the form needed for a URL or URI, you have two choices, depending upon whether or not you want the line-of-business name to be part of the Internet domain name or the path. If you want it to be part of the Internet domain name, you end up with

```
http://insurance.nouveau.com  
http://pharmacy.nouveau.com
```

If you want the line of business to be part of the path, you end up with

```
http://nouveau.com/insurance  
http://nouveau.com/pharmacy
```

This, in practice, may not be an arbitrary choice: Making the line of business part of the Internet domain name allows distinct machines to be used as targets for the different lines of business; making the line of business part of the path forces a single machine to be used as the entry

point for both lines of business. There may be organizational and management issues involved. These are discussed later in this chapter in the Complicating Realities section.

To deal with this type of situation, many enterprises employ network appliances that can make seamless conversions between these two formats. Although this adds flexibility, it requires network configuration—yet another administrative task.

Best Practice: Global Uniqueness

Include an enterprise-specific qualifier to the functional context. If the enterprise has independent lines of business, include the line of business in the functional context as well.

A Notational Convention for Internet Domain Names

This question of how much of the idealized name will become part of the Internet domain name occurs so often that we will adopt a convention for indicating the answer: The portion of the idealized structure that will become the Internet domain name will be outlined with a border. Using this convention, the example of the line-of-business name being part of the Internet domain name would be idealized as

```
<enterpriseID><lineOfBusiness><functionalArea>
```

The example of the line-of-business name being part of the path would be idealized as

```
<enterpriseID><lineOfBusiness><functionalArea>
```

WSDL and XSD Namespace URIs

The idealized structure for a WSDL namespace URI has the form

```
<functionalContext><abstractService><abstractInterface>wsdl
  <wsdlName>
```

Expanding the functional context, we have

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <abstractInterface>wsdl<wsdlName>
```

It is common practice to make the actual name structure a URI. For the Payment Manager service's Claim Payment interface WSDL and including the major version number in the name, this gives

```
http://insurance.nouveau.com/finance/paymentManager/
  claimPayment/wsdl/claimPayment1
```

The idealized structure for a schema namespace URI has the form

```
<functionalContext><abstractService><abstractInterface>schema
  <schemaName>
```

Expanding the functional context, we have

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
<abstractInterface>schema<schemaName>
```

It is common practice to make the actual name structure a URI. For the Claim Payment schema associated with the Claim Payment WSDL and including the schema major version number, this gives

```
http://insurance.nouveau.com/finance/paymentManager/
  claimPayment/schema/claimPayment1
```

Best Practice: WSDL and Schema Namespace URIs

Include the full functional context, abstract service name, abstract interface name, and WSDL or schema name in the WSDL and schema namespace URIs. Insert `wsdl` or `schema` prior to the WSDL or schema name to differentiate WSDL and schema namespaces.

WSDL and XSD Filenames

It is a good practice is to use the WSDL or schema namespace structure as the basis for defining the structure of the filename (i.e., the path to the file). The idea is to make the fully qualified filename a URL (or at least a URI). Making it a URL tells you where to locate the file. With this approach, the idealized structure for a filename is

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <abstractInterface>wsdl<filename>
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <abstractInterface>schema<filename>
```

In the implementation, there are a couple of options depending on how the systems hosting the files are physically organized. One uses a single machine as the access point for all files:

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>  
<interface><filename>
```

Preferable is an approach that uses a different machine per line of business:

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>  
<interface><filename>
```

This yields the following type of URL:

```
http://insurance.nouveau.com/finance/paymentManager/  
claimPayment/wsd1/ClaimPayment.wsd1
```

In some situations, the file will be present in a file system. In such cases, the full structure of the name should be preserved in the folder hierarchy of the file system. For example, you might have

```
file://<rootPath>/com/nouveau/insurance/finance/paymentManager/  
claimPayment/wsd1/claimPayment.wsd1
```

This, however, can lead to deep folder structures in which there is only a single folder in each of the upper-level folders. When converted to Java class structures, it also leads to a Java class for each level. In such cases, the upper part of the structure can be collapsed into a single folder name:

```
file://<rootPath>/com.nouveau.insurance.finance/paymentManager/  
claimPayment/wsd1/claimPayment.wsd1
```

Best Practice: WSDL and Schema Filenames

Include the full functional context, abstract service name, interface name, and wsdl or schema in the WSDL and schema filenames.

WSDL Names

To ensure the uniqueness of the WSDL name and to make it clear which WSDL it is, it is a good practice to make the WSDL name the same as the fully qualified filename, which should be a URI or URL.

```
http://insurance.nouveau.com/finance/paymentManager/  
claimPayment/wsd1/claimPayment.wsd1
```

Schema Locations

The schema location, by definition, is a URI and is ideally a URL. If the above naming practice for schema filenames is followed, then the fully qualified filename is the schema location.

```
http://insurance.nouveau.com/finance/paymentManager/  
claimPayment/schema/claimPayment.xsd
```

WSDL-Specific Schema Location

Each WSDL requires a schema that defines the types used in its message definitions. The WSDL standard provides two options for supplying the schema: importing a schema file or placing the schema definitions in-line within the WSDL file. The latter approach is preferred. This is because an `import` statement in the WSDL requires retrieving the file, which requires access to that file. In many environments, providing access to these files is impractical. For this reason it is a best practice to place the WSDL schema definitions directly in the WSDL file rather than using an `import` statement.

Best Practice: WSDL-Specific Schema Location

Place WSDL schema definitions in the WSDL file.

WSDL Message Names

When defining WSDL messages, there are two different namespaces involved, one for the message itself and the other for the schema that defines the element type used in the message. While technically the two namespace URIs are allowed to be the same, it creates a very confusing situation when looking at the WSDL. For this reason, it is good practice to include WSDL or schema as the last field in the namespace URIs to keep them distinct, which is the earlier-recommended best practice.

The messages themselves are defined within the WSDL namespace. Typically each message is intended to play a particular role with respect to a specific operation, so it is good practice to include the `<operationName>` as part of the message name. This gives you the following idealized structure for the fully qualified message name:

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>  
<interfaceName><operationName><messageName>
```


There is a wrinkle in turning this idealized structure. The name given to the message in the WSDL file is defined directly in the context of the WSDL namespace, not under the operation. Thus if two operations have the same message name (such as `requestMessage` or `responseMessage`), there will be a conflict. To avoid this, you simply concatenate the operation name with the message name to give the implementation, for example, `PayClaimRequestMessage`. For the claim payment operation, this gives us the following message definitions in the WSDL.

```
<message name = "PayClaimRequestMessage">
  <part name = "request" type = "ns:PayClaimRequest"/>
</message>
<message name = "PayClaimResponseMessage">
  <part name = "response" type = "ns:PayClaimResponse"/>
</message> <message name="processShipmentRequest">
```

Each message has parts, and each part has a name that is also defined in the WSDL namespace. Since the scope of the part name is local to the message definition, the actual name is not particularly important.

Each part requires a type definition. The type refers to a datatype that is defined in the schema namespace referenced by the WSDL.

Port Type, Service, and Binding Names

An idealized structure for these names has the form

```
<functionalContext><abstractService><interfaceName>
  <portTypeName>
</functionalContext><abstractService><interfaceName>
  <serviceName>
</functionalContext><abstractService><interfaceName>
  <bindingName>
```

Note that, following the best practice recommendations, there is some redundancy here. The best practice is to have a WSDL for each interface and to include the interface name in the WSDL namespace URI. The `portType` is actually the interface, so appending it to the namespace name is redundant. However, since the `portType` is actually defined within this namespace, this is the structure you actually get. A similar situation arises for the service and binding names.

Using the `portType` name as an example and expanding the functional context, you have

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <interfaceName><portTypeName>
```

In practice, this idealized structure never appears as a single string in the WSDL. Instead, it is divided into two parts. The first part of the structure is represented by the target namespace in the WSDL. By the best practice recommendation, this includes everything up to and including the interface name.

The second part is the `portType` name, which is declared within the scope of the namespace. The names of the `service` and `binding` are similarly declared within the namespace. Here are some example fragments from the `claimPayment.wsdl` file:

```
<definitions
  xmlns:tns=http://insurance.nouveau.com/finance/
    paymentManager/claimPayment/wsd1/claimPayment1
  targetNamespace="http://insurance.nouveau.com/finance/
    paymentManager/claimPayment/wsd1/claimPayment1
  xmlns:ns="http://insurance.nouveau.com/finance/
    paymentManager/claimPayment/schema/claimPayment1">
...
  <portType name="ClaimPayment">
    <operation name="payClaim">
      <input message="tns:PayClaimRequestMessage"
        name="input"/>
      <output message="tns:PayClaimResponseMessage"
        name="output"/>
    </operation>
    ...
  </portType>

  <service name="ClaimPaymentService">
    <port name="ClaimPaymentPort"
      binding="tns:ClaimPaymentPortBinding">
      <soap:address location="http://insurance.nouveau.com:5555/
        com.nouveau.insurance.finance.paymentManager.
        claimPayment.endpoint1"/>
    </port>
  </service>

  <binding name="ClaimPaymentPortBinding" type="tns:ClaimPayment">
    <soap:binding style="document" transport="http://
      schemas.xmlsoap.org/soap/http"/>
    <operation name="payClaim">
      <soap:operation style="document" soapAction="/
        com.nouveau.insurance.finance.paymentManager.
        claimPayment1.payClaim"/>
    <input>
      <soap:body use="literal" parts="request"/>
    </input>
    <output>
```

```

        <soap:body use="literal" parts="response" />
    </output>
</operation>
</binding>
</definitions>

```

There is a nuance here worth pointing out: portTypes are not directly associated with services. Instead, a declaration known as a *binding* provides details of the portType's implementation, and then that binding is used in defining a port (an actual interface instance or endpoint) on the service. The port has its own name. For clarity, it is good practice to choose names for ports and portTypes that distinguish between the two.

Operation Names

An idealized structure for operation names has the form

```

<functionalContext><abstractService><interfaceName>
    <operationName>

```

Expanding the functional context, you have

```

<enterpriseID><lineOfBusiness><functionalArea><abstractService>
    <interfaceName><operationName>

```

In practice, this idealized structure never appears as a single string in the WSDL. Instead, it is divided into two parts. The first part of the structure is represented by the target namespace in the WSDL. By the best practice recommendation, this includes everything up to and including the interface name. The second part is the operation name, which is declared within the scope of the portType (interface).

SOAP Address Location

The SOAP address location is a URI that indicates the connection point (physical destination) to which operation requests are directed. Depending upon the policy you choose for separating the traffic directed to different services, you should use one of the following idealized structures for this URI:

```

<enterpriseID><lineOfBusiness><functionalArea>
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
    <interfaceName>

```

The choice between these is not arbitrary. If the first form is used, all requests for a given functional area will be sent to the same location.

The second form offers at least the possibility that requests for different services may be directed to different components, and the third can distinguish based on the interface.

This difference may have significant performance implications for the subsequent implementation, particularly if one interface provides real-time operations involving small data structures and quick response while another provides batch operations involving huge data structures and asynchronous responses. Performance considerations drive you toward separating this traffic, but separation requires sufficient information in the location.

Another consideration is that, for many implementation technologies, the use of a detailed path allows the convenient routing of requests for different services and interfaces.

WSDL 2.0 allows multiple endpoints for an interface, giving yet another possibility:

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <interfaceName><endpointName>
```

It is generally a good idea to include too much rather than too little information in the location. It is relatively easy to ignore the lower-level details, but you cannot synthesize information that is not present.

For JMS bindings, the idealized structure translates in a very straightforward way into JMS destination names:

```
enterpriseID.lineOfBusiness.functionalArea
enterpriseID.lineOfBusiness.functionalArea.serviceName
enterpriseID.lineOfBusiness.functionalArea.serviceName.
  interfaceName
enterpriseID.lineOfBusiness.functionalArea.serviceName.
  interfaceName.endpointName3
```

With HTTP bindings a further implementation consideration, relevant to locations, is the choice of a socket number. The structure of a URL is determined by the transport that has been selected. For http (and https) transports, the syntax is

```
http://<hostIdentifier>:<socketNumber>/<path>
```

This raises the question as to which parts of the idealized namespace should map to the `<hostIdentifier>` and which parts should map

3. A fully qualified destination name provides maximum flexibility in assigning destinations to JMS servers.

to the `<path>`. Answering this question requires the recognition that the combination `<hostIdentifier>:<portNumber>` denotes a physical destination, while the `<path>` is used logically at that destination to further distinguish between requests. There are architectural trade-offs involved in this decision, with considerations that go beyond the scope of this book (e.g., how many sockets should there be versus what load can each socket reasonably handle).

For the purposes of the discussion here, consider the socket-Number to be part of the Internet domain name. Assuming you want to include the structural detail down to the interface level, this gives you the following possibilities for the idealized structure of the location URL:

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <interfaceName>
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <interfaceName>
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <interfaceName>
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <interfaceName>
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <interfaceName>
```

The first indicates that a single socket will handle all of the service requests for the entire enterprise. This requires centralized management of the port and a common technology touch point for all services. Generally, the technology employed here uses the `<path>` to redirect different requests to different sockets. The second results in a single socket for each line of business, which again may employ technology to redirect requests to different sockets. The third and fourth choices generally reflect the actual structure of the implementations to which requests of the first or second type are redirected. If you have multiple endpoints per interface, you will need to append an `<endpoint>` field to each of these possibilities.

Best Practice: Location Names

Include the full functional context, abstract service, and interface in the location name.

soapAction Names

The `soapAction` name is nearly identical to the fully qualified operation name as described earlier, with the major version number appended to the interface name. An idealized structure for `soapAction` names has the form

```
<functionalContext><abstractService><interfaceName+major>
  <operationName>
```

Expanding the functional context, you have:

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <interfaceName+major><operationName>
```

This approach provides the maximum flexibility in defining endpoints. Using this approach, all requests for all operations could be sent to a single endpoint (location) without ambiguity: The `soapAction` uniquely identifies the required operation.

Best Practice: SOAP Action Names

The SOAP action name should be the fully qualified operation name with the major version number appended to the interface name.

Schema Types Specific to an Interface and an Operation

As mentioned in the previous section, schema types are often created that are dedicated to a single operation. The ideal structure for an operation-specific message type name would be

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <interfaceName><operationName><typeName>
```

This leads to idealized names like

```
com.nouveau.insurance.finance.paymentManager.claimPayment.
  payClaim.Request
com.nouveau.insurance.finance.paymentManager.claimPayment.
  payClaim.Response
```

In reducing this idealized structure to an implementation you encounter a limitation of the schema definition language: It does not support hierarchical names within the namespace. Specifically, the `Request` and `Response` cannot be scoped within the `payClaim` operation

name—they have to be declared directly within the namespace. Thus if other operations have `Request` and `Response` data types (as they likely would), then you'll have a name conflict. The solution to this problem is to concatenate the operation name and the ideal message name, giving type names like `PayClaimRequest` and `PayClaimResponse`.

These definitions would occur (following the earlier recommended best practice) within the schema namespace:

```
http://insurace.nouveau.com/finance/paymentManager/  
claimPaymentInterface/schema/claimPaymentInterface1
```

Schema Shared Data Types (Common Data Model Types)

Even though the types that represent entire messages are often specialized for the operations they support, the subordinate data types that they reference many times can be shared between messages. If these shared data types are specific to the interface, then they can remain in the interface's namespace. But if they are of more general intent, they belong in namespaces of their own.

You might, for example, want to define the standard representation for an address as shown in Figure 8-4. You might want to standardize the use of this data type across all of Nouveau Health Care. To do so, you use the namespace in which the type is defined to indicate the intended scope of utilization.

Types like `Address Type` often have other closely-related types, such as the `ISO Country Code`. To keep related concepts grouped together yet separated from other concepts, it is a good practice to define a namespace for the concept and its related elements, which shall be termed a

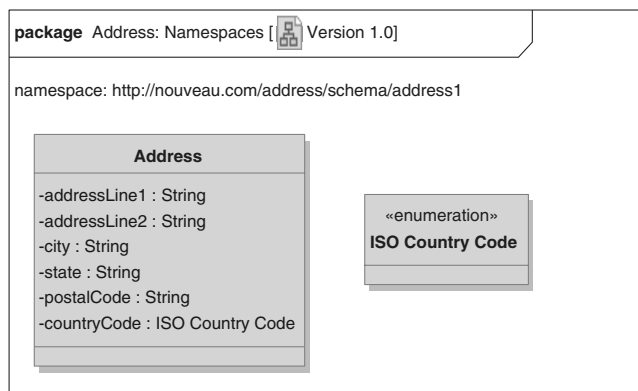


Figure 8-4: *Address Type*

concept package. In this example, the concept package is named **Address** and, in keeping with the versioning best practices, has the version major number appended.

The portion of the namespace that precedes the package name is determined by the intended scope of utilization. This gives the following possibilities for the fully qualified type name:

```
<enterpriseID><conceptPackage>schema<schemaName>
<enterpriseID><lineOfBusiness><conceptPackage>
  schema<schemaName>
<enterpriseID><lineOfBusiness><functionalArea><conceptPackage>
  schema<schemaName>
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <conceptPackage>schema<schemaName>
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
  <abstractInterfaceName><conceptPackage>schema<schemaName>
```

Schema names should be versioned in the same manner as interfaces, with **<major>** and **<minor>** version numbers. Wherever the **<schemaName>** name appears in a name structure, its major version should be appended. This includes the namespace URI for the schema and the XSD filename. Note that the XSD filename also includes the minor version number.

Complicating Realities

Technology Constraints

Reality often constrains the ideal approach. This is particularly true with names. Ideally, names have no restrictions with respect to either the number of fields or the length of any individual field. Unfortunately, this does not hold true in practice. The various technologies being employed have both theoretical restrictions and practical limitations.

Internet domain names limit the number of fields in a name to 127 and allow each field to contain up to 63 octets (bytes). The whole domain name (including “.” separators) is limited to 253 octets. In practice, some domain registries may have shorter limits.

The JMS standard places no limitations on names, but vendor implementations often do. For example, Enterprise Messages Service destinations can have up to 64 fields. Individual fields cannot exceed 127 characters. Destination names are limited to a total length of 249 characters.

Whatever naming standards are established, they must obviously take into consideration the constraints of the chosen supporting technologies.

There is a practical limitation to consider as well. By default, every byte (octet) of every name must be transmitted with every message. Thus it is a good practice to avoid excessive length in names, while at the same time preserving the readability of the name.

Naming Authorities

Every position in a naming hierarchy requires an authority for its administration. As discussed earlier, each naming authority (a) directly manages the field level for which it is responsible, and (b) establishes the policy for managing the lower-level fields.

The ability to delegate the management of the lower-level structures is part of the power of hierarchical name structures. At the enterprise level, the authority must

- Assign names to represent each of the lines of business.
- Identify who in each line of business will manage the lower-level structures below the line-of-business field.
- Assign names to represent each enterprise-level service.
- Identify who will manage the lower-level structure for each enterprise service.
- Assign names to represent each enterprise-level concept package
- Identify who will manage the lower-level structure for each enterprise concept package.

Within each line of business, the responsibilities are similar: assigning names for line-of-business services, functional areas, and concept packages, and identifying who will manage the lower-level structures for each.

Complex Organizational Structures

Functional Organizations

If a line-of-business organization is functionally complete (i.e., horizontally integrated), then all of the services and concepts required to operate the line of business are part of that organization. The naming structure we have been discussing reflects the line-of-business's singu-

lar line of authority over all the services and concepts relevant to the organization.

Many organizations, however, are structured functionally rather than by line of business. An organization might have functional groups for marketing, sales, logistics, finance, engineering, manufacturing, and so forth. If such an organization provides capabilities uniformly across multiple lines of business, the top-level structure may be more appropriately based on function rather than line of business:

```
<enterpriseID><functionalArea>
```

Mixed Functional and Line-of-Business Organizations

Some organizations combine these two approaches. A line-of-business organization may well have a functional substructure under each line of business. In such cases, the structure of the functional context must be expanded. This gives a functional context of

```
<enterpriseID><lineOfBusiness><functionalArea>
```

Less frequently it may be appropriate to reverse the hierarchy:

```
<enterpriseID><functionalArea><lineOfBusiness>
```

The challenge here is that there are, in reality, two hierarchies: the functional hierarchy and the line-of-business hierarchy. Yet the representational technologies (WSDL and schema) only allow for one hierarchy, so the two must be combined into a single functional context. This forces one to be somewhat arbitrarily chosen to dominate the other.

Regardless of how the hierarchies are combined, some caution is in order: Enterprises tend to reorganize. Since you are going to be living with the namespace structure even after the reorganization, it is important to choose a structure and names for both functions and lines of business that will remain stable over time. Bear in mind that a single organization may actually serve more than one functional purpose (e.g., sales and marketing could be in one parent organization) or serve more than one line-of-business purpose (e.g., a division that handles multiple lines of business). In such cases, each individual purpose should have a distinct name in the namespace. This makes the namespace structure relatively stable with respect to reorganization.

Geographic Distribution

Another challenge you are liable to encounter is geographic distribution. This is particularly true for multinational enterprises, where the

operations in different countries or regions tend to be conducted by legally distinct entities. This adds yet a third hierarchy to the ways that the enterprise can be viewed (Figure 8-5).

The presence of yet a third hierarchy further complicates the functional context. One possible structure that could emerge is

```
<enterpriseID><geographicRegion><lineOfBusiness><functionalArea>
```

This is but one of six possible combinations, and that does not even count the possibilities of leaving one or more of these hierarchies out of the structure entirely. Altogether, there are 15 possibilities. So how do you select the one for your enterprise?

There are two factors that should drive your thinking. The first one focuses on organizational realities. In this you need to consider the following:

- What does the current organizational structure look like (i.e., what is the actual organizational hierarchy today)?
- What are the realistic prospects for cooperation between organizations?
- Is there a place in the parent organization for a working group to manage namespaces that cross organizational boundaries?
- Is there sufficient authority for such a working group to effectively manage such a namespace (i.e., will the other organizations listen)?

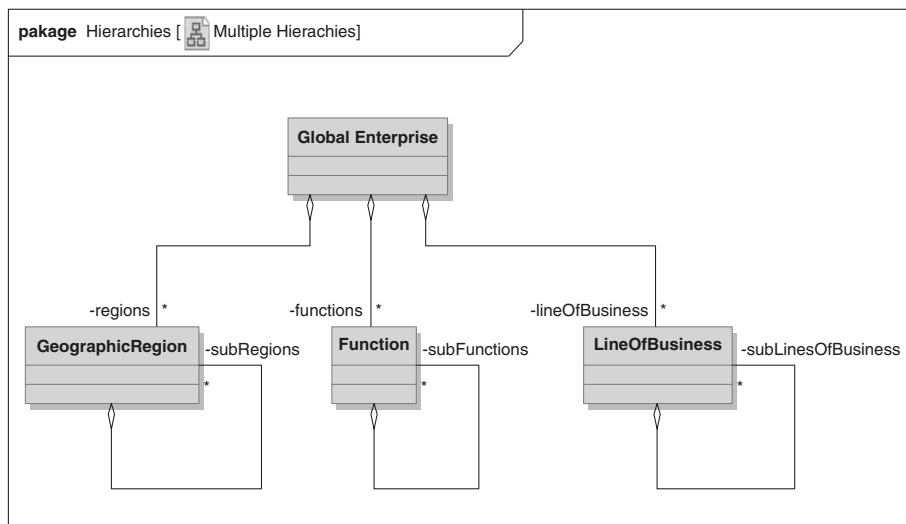


Figure 8-5: *Multiple Hierarchies in the Enterprise*

If the prospects look dim for cooperation between organizational units, then you have little choice other than to adopt a namespace structure that treats the organizations as being independent (see naming authorities earlier).

The other factor to consider is the intended future direction for the enterprise in terms of sharing data, resources, services, and business processes across the organization. If there is an effort underway to increase sharing (which presupposes that there is a reasonable prospect for cooperation between organizations), then the hierarchy you choose should be one that makes the most sense with respect to the chosen direction.

Environments: Values for Address Location Variables

Each service will, in its lifetime, exist in a number of environments, ranging from development through several test environments and, finally, one or more production environments. Thus it becomes necessary to distinguish among the different instances of the service when accessing the service.

The logical place to make this distinction would be in the address location in the WSDL. Unfortunately, the structure of the WSDL does not make provisions for designating different addresses for different environments. Even worse, changing the WSDL to indicate the different environment requires changes to both the consumer and the provider. From a change management and versioning perspective, since the WSDL is the service interface definition, edits to the WSDL could potentially change anything. Therefore, editing the WSDL invalidates whatever testing has been performed and thus defeats the purpose of the WSDL as a specification.

To work around this limitation, most service consumer and service provider implementation technologies allow the address (endpoint) to be provided by a variable whose value is set at deployment time. With this approach, environments must be distinguished within the values provided for the address. These addresses (the provided values) need to be distinct from one another while remaining descriptive of their purpose so that they can be accurately and appropriately set at deployment time.

Referring back to the earlier discussion of addresses, a portion of the idealized address structure maps to the `hostIdentifier`. This is the level at which development, test, and production environments should be distinguished from one another. Using the scheme described

earlier, you have the following possibilities (the portions in the box correspond to the host identifier):

```

<enterpriseID><environment><lineOfBusiness><functionalArea>
  <abstractService><interface>
<enterpriseID><lineOfBusiness><environment><functionalArea>
  <abstractService><interface>
<enterpriseID><lineOfBusiness><abstractService><environment>
  <functionalArea><interface>
<enterpriseID><lineOfBusiness><abstractService><functionalArea>
  <interface><environment>
<enterpriseID><lineOfBusiness><abstractService><functionalArea>
  <interface><environment>

```

For HTTP bindings, manifestation of the `<environment>` can take one of two forms: It can be the `hostIdentifier` itself or the socket on the host. *It is generally not a good practice to use the same `hostIdentifier` and socket for multiple environments. It is generally not a good practice to share a `hostIdentifier` between a production environment and any other environment.*

For JMS bindings, the manifestation of the `<environment>` can also take one of two forms: It can be the `hostIdentifier` and socket for the JMS server itself or it can be included directly in the queue name. *It is generally not a good practice to share a JMS server between a production environment and any other environment.*

Deployment Flexibility for HA and DR

An address (at least in the `http[s]` type of addressing) identifies a specific machine to which requests are to be sent. Should this machine move (for fault tolerance, site disaster recovery, or simply administrative convenience), all the interfaces accessed through that machine would become unavailable. One way around this is to use virtual IP addresses or hostnames, allowing the networking infrastructure to reroute requests to another machine. Another approach is to make the address actually an `http` query with a response that is the real address. Amazon e-commerce web services, for example, use this approach:

```

<soap:address location="https://ecs.amazonaws.com/onca/
  soap?Service=AWSECommerceService"/>

```

Abstracting the logical structure here, the `<hostIdentifier>/<path>` part of the location actually identifies the server to which the query is being submitted, while the value of the `Service` parameter indicates

the desired service. This approach facilitates the flexible rehosting of the actual service providers over time.

For greatest flexibility, it is recommended that the full path name of the interface be used as the argument for this type of query. Thus if one query service is being used to support the entire enterprise, the value for the query would have the structure

```
<lineOfBusiness><abstractService><interfaceName>
```

If the query service supports just a line of business, then the query value would be

```
<abstractService><interfaceName>
```

It is not good practice to use the same query service for different environments.

SOAP over JMS utilizes JNDI lookups in a similar manner. The same considerations apply.

Developing Your Standard

To begin with, you need to identify the organization that will have overall responsibility for the naming standards in your enterprise. If you are in the early stages of adopting SOA, you may not be in a position to mandate that a particular naming structure be followed. Nevertheless, there is no reason that you cannot adopt a structure that can be appropriately generalized to meet the needs of the larger enterprise as outlined in this chapter.

The enterprise-level authority has several responsibilities. One is to define an overall strategy for

- WSDL and schema namespace URIs, including versioning
- Location names, including:
 - Versioning
 - Directing requests to the appropriate environment
 - Using indirection (http query or JNDI lookup) for addresses

Another responsibility is to identify which organizations will be responsible for managing the lower levels of the naming structure. If the enterprise is large, this will likely mean identifying an organization in each line of business that is responsible for names used in that line of business. Each of those organizations, in turn, will have to determine

how much of the naming structure it will directly manage and how much management responsibility it will delegate to other organizations.

Summary

In the complex world of IT, how things are named has a significant influence on the usability of components when building solutions. Clear, descriptive names make it possible for you to identify things without having to look them up. The structure of the name also helps to avoid ambiguity and name conflicts.

There are a number of general principles that should guide the creation of names. Names should have a hierarchical structure. That hierarchical structure should follow a general-to-specific organization. The names chosen should remain stable over time. With the exception of enterprise identifiers, the names used to identify organizations and systems should be avoided and generic names used instead. The use of acronyms should be avoided unless they are universally understood. Type and instance names should be clearly distinguishable. Conventions should be adopted to ensure that multi-word fields are easily readable. For ease in managing versions, each interface should have its own WSDL with its own namespace.

A good idealized name structure has the form

```
<functionalContext><abstractService><abstractInterface>
```

The functional context commonly expands as follows, with the presence of the `<enterpriseID>` avoiding naming conflicts in the event of mergers and acquisitions:

```
<functionalContext> ::= <enterpriseID><lineOfBusiness>
    <functionalArea>
```

The `<abstractService>` may encompass multiple interfaces, each with its own WSDL. For service operations, the `<abstractInterface>` commonly expands to

```
<abstractInterface> ::= <interfaceName><operationName>
```

Ideally, WSDL namespaces, WSDL filenames, and SOAP addresses (endpoints) should have the idealized form

```
<enterpriseID><lineOfBusiness><functionalArea><abstractService>
    <interfaceName>
```

SOAP action names should have this form with the operation name appended at the end. Schema namespaces and filenames should use the portion of this structure that describes the intended scope of utilization of the schema.

Technology constraints on the length of individual fields and the overall length of identifiers must be taken into consideration when defining names. The enterprise should establish a policy defining the hierarchical structure of names to be used and identifying the organization responsible for establishing the values of the top-level fields and the organizations responsible for ensuring the uniqueness of subordinate field values.

Index

A

- Abbreviations, avoiding in naming structure, 188
- Abstract architecture pattern
 - overview of, 8
 - reference architectures as, 13
 - showing dependencies with, 112
- Abstract definitions, WSDL, 176–177
- Abstract, service
 - overview of, 122–123
 - service specification contents, 124
 - service specification documentation, 420
 - service specification example, 128
- Abstract services, 174–175, 178–180
- Accept Queue Size parameter, HTTP Connector, 345
- Acceptor threads, HTTP Connector, 345–347
- Access control, service federation, 402–403, 405–406
- Access points, 143, 405–406
- Access rate, benchmark results, 332–334
- Accounts and fund transfers, Payment Manager, 77–81
- Acknowledge Mode parameter, JMS process starters, 360–361
- Acronyms, avoiding in naming structure, 188
- Actions, Hawk rules, 51–52
- Actions, UML activity diagrams, 437–439
- Activation Limit tuning parameter, ActiveMatrix BusinessWorks, 367–368
- ActiveMatrix BusinessWorks. *See* TIBCO ActiveMatrix BusinessWorks™
- ActiveMatrix deployment options, 31–33
- ActiveMatrix Adapters. *See* TIBCO ActiveMatrix® Adapters
- ActiveMatrix Administrator. *See* TIBCO ActiveMatrix® Administrator
- ActiveMatrix composite implementation, 307–308
- ActiveMatrix® Decisions. *See* TIBCO ActiveMatrix® Decisions
- ActiveMatrix Service Bus failover, 390–391
- ActiveMatrix Lifecycle Governance Framework. *See* TIBCO ActiveMatrix® Lifecycle Governance Framework
- ActiveMatrix Product Portfolio. *See* TIBCO ActiveMatrix® Product Portfolio
- ActiveMatrix Service Bus. *See* TIBCO ActiveMatrix® Service Bus
- ActiveMatrix Service Grid. *See* TIBCO ActiveMatrix® Service Grid
- Activity diagrams, UML
 - documenting process-pattern mapping, 7
 - documenting triggered behaviors, 113
 - notation reference for, 437–440
- Adapters. *See also* TIBCO ActiveMatrix® Adapters
 - external access interaction via, 38–39
 - Hawk, 53–54, 56
 - single-threaded solutions with, 276
 - TIBCO Adapter for Files, 62–63
 - TIBCO Adapter for IBM i, 61–62
- Address location
 - SOAP, 200–202
 - values for variables in, 209–210
 - versioning SOAP endpoints, 168
- Address schema, WSDL Sales Order example, 444–445
- AeRvMsg format, BusinessConnect, 64
- Aggregation, UML Class Diagram notation, 430–431
- Alerts, Hawk Event Service, 55
- AMI (Application Microagent Interface)
 - interfaces, Hawk Agent, 56
- Appendices
 - service specification documentation, 423
 - solution architecture documentation, 417
- Applications, composite, 304
- Architecting BPM Solutions with TIBCO®* (forthcoming, TIBCO Press and Addison-Wesley), 426
- Architecting Complex Event Processing Solutions with TIBCO®* (forthcoming, TIBCO Press and Addison-Wesley), 425

- Architecture fundamentals, TIBCO
 - ActiveMatrix Adapters, 30–31
 - ActiveMatrix BusinessWorks, 28–30
 - ActiveMatrix deployment options, 31–33
 - ActiveMatrix Product Portfolio, 27
 - ActiveMatrix Service Bus, 27–28
 - ActiveMatrix Service Bus policies, 44–45
 - ActiveMatrix Service Grid, 28–29
 - design patterns, 33–44
 - Enterprise Message Service (EMS)
 - product, 26
 - overview of, 25
 - summary review, 46
- Architecture pattern
 - as architecture view, 6
 - in composite service architecture, 20–21
 - documenting service, 145–146
 - documenting test harness, 317–319
 - hierarchy of architecture and, 7–11
 - Nouveau Health Care case study, 70
 - In-Only, 33–34, 102–104
 - Out-In, 34
 - Out-Only, 34
 - process-pattern mapping executing, 6–7
 - service utilization, 18
 - showing dependencies with abstracted, 112
 - solution architecture, 14–16
 - solution architecture documentation of, 411
 - used in this book. *See* Design patterns, used in this book
 - for versioning, 165–167
- Architectures
 - component life cycle and, 22–23
 - composite service, 20–22
 - design pattern, 13
 - hierarchy of, 7–11
 - making distinctions in, 11–12
 - service, 17
 - service utilization contract and, 22
 - service utilization pattern, 17–19
 - solution, 14–16
 - summary review, 23–24
 - views, 4–7
- Artifacts, UML activity diagrams, 437
- Associations
 - designing XML schema by identifying, 228–229
 - representing in data models, 226–227
 - UML Class Diagram notation for, 429–430
- Asynchronous coordination patterns, 261–262
- Asynchronous events, choreography with, 298–299
- Asynchronous JMS request-reply interactions, 257–261
- Asynchronous request-reply coordination pattern, 40–41, 268
- Asynchronous storage replication strategies, site failover, 377–378
- Augmentation and content transformation mediation pattern, 36–37
- Autostart Core Threads parameter, named thread pools, 344
- Availability. *See also* FT and HA (fault tolerance and high availability)
 - defined, 372
 - service usage contract specifying, 144
 - solution architecture documentation, 412
- B**
- B2B (business-to-business) protocols, BusinessConnect, 63–64
- Back-end systems
 - accessing, 243–244
 - solution architecture decisions, 240–243
- Backwards compatible changes
 - architecture pattern for, 165–166
 - defined, 152
 - OSGi version numbering scheme for, 153–154
 - versioning SOAP interface addresses and, 168
 - to WSDL and XML schema, 160–163
- Basic federation pattern, 403–405
- Basic interaction design patterns, 33–34
- Bathtub (or fire hose) test, benchmark results, 314–315
- Behaviors. *See also* Observable dependencies and behaviors
 - composite, 293–294
 - defining component, 95–96
 - documenting nonfunctional, 116–117, 127, 140
 - documenting service architecture, 145–149
 - managing composite, 296–298
 - observable. *See* Observable dependencies and behaviors
 - solution architecture documentation, 416–417
 - triggered. *See* Triggered behaviors
- Benchmarking
 - of complex components, 324–327
 - determining operating capacity in, 316–317
 - identifying capacity limit, 327–328
 - identifying CPU utilization limits, 328–329
 - identifying disk performance limits, 331–334
 - identifying memory limits, 335–336

- identifying network bandwidth limits, 329–331
 - identifying test harness limits, 336
 - misleading results of, 314–316
 - objectives, 313
 - summary review, 338–339
 - using results of, 326–327
 - Benchmarking, documenting test design
 - experimental parameters, 321–322
 - overview of, 317
 - test harness architecture pattern, 317–318
 - test harness process mapping, 318–321
 - test results, 322–324
 - Benefits Service, Nouveau Health Care case study, 80–81
 - Binding names, WSDL, 198–200
 - Black box perspective
 - composites not suitable for, 117–118
 - of observable dependencies and behaviors, 96–97
 - BPMN diagrams, 7
 - Brackets, OSGi versioning ranges, 155
 - Bug fixes, 154–156
 - Building-block design patterns
 - asynchronous JMS request-reply interactions of, 257–261
 - business exceptions, 252–257
 - for dual coordination patterns, 261–262
 - objectives, 239
 - rule service governing process flow, 244–250
 - rule services and data, 250–252
 - separating interface and business logic, 240–243
 - solution architecture decisions, 240
 - summary review, 262–264
 - using services for back-end systems, 243–244
 - Business logic, solution architecture decisions, 240–243
 - Business processes
 - managing, 425–426
 - managing with rule service. *See* Rule service, process flow
 - naming based on stable concepts for, 187
 - Nouveau Health Care case study, 68–69
 - Sales Order Service in Order-to-Delivery. *See* Sales Order Service example
 - solution architecture, 14–16
 - solution architecture documentation, 409–414
 - Business requirements, nonfunctional behavior, 116–117
 - Business-to-business (B2B) protocols, BusinessConnect, 63–64
 - Business variations, expected, 252–257
 - BusinessConnect. *See* TIBCO BusinessConnect™
 - BusinessConnect DMZ Component, 64
 - BusinessConnect Interior Engine, 63–64
 - BusinessEvents. *See* TIBCO BusinessEvents®
 - BusinessWorks. *See* TIBCO ActiveMatrix BusinessWorks™; TIBCO ActiveMatrix BusinessWorks™ Service Engine
 - BusinessWorks SmartMapper. *See* TIBCO BusinessWorks™ SmartMapper
- ## C
- Cache
 - avoiding with nested retrieval, 111
 - observable state and, 108–111
 - speeding up lookup process, 306
 - System-of-Record-with-Cached-Read-Only-Copies pattern, 285–286
 - Cached Information Pattern, composite services, 304–306
 - Call Operation Action, UML activity diagrams, 438
 - Capacity limit, benchmarking, 327–329
 - Capacity measurement tests, 314
 - Capitalization, structured name design, 189–190
 - Carrier schema, WSDL Sales Order example, 445–446
 - Cascading Control Pattern, composite services, 304
 - Case study. *See* Nouveau Health Care, case study
 - Checkpoints, ActiveMatrix BusinessWorks failover, 387
 - Choreography design pattern, 297–299
 - Claim Payment Service Implementation, Payment Manager Composite, 296
 - Claims management, case study
 - architecture pattern, 70–71
 - payment domain concepts, 80–83
 - Payment Manager service interfaces for, 129–132
 - Class Diagram. *See* UML Class Diagram notation
 - Classes, UML
 - associations between, 429–430
 - Class Diagram notation, 427–428
 - part-whole relationship between two, 430–431
 - “Classic,” TIBCO, 29
 - Client Acknowledgement Mode, JMS process starters, 360–361
 - Client connection load distribution, EMS, 269–271, 393–396
 - Client libraries, EMS, 26
 - Client, providing data to rule service, 250–251

- Cluster failover pattern, 388–390
- Collaborations, UML notation, 440
- Combined Pattern, TIBCO EMS client load distribution, 271
- Command Center, TIBCO Managed File Transfer, 58
- Command-line interface, configuring EMS servers, 26
- Commercial off-the shelf packages. *See* COTS (commercial off-the shelf) software packages
- Communications latency, documenting test process mapping, 320
- Compatibility, and dependencies, 152
- Compensating transactions, 44–45
- Complex components, in benchmarking, 324–327
- Components
 - ActiveMatrix Adapters, 30–31
 - ActiveMatrix BusinessWorks, 28–30
 - ActiveMatrix Service Bus, 27–28
 - ActiveMatrix Service Grid, 28–29
 - architectural distinctions and, 11–12
 - benchmarking complex, 324–327
 - in composite service architecture, 20–22, 303
 - documenting under test parameters, 321
 - implementation architecture for, 9, 11
 - life cycle, 22–23
 - misleading benchmark results, 315
 - monitoring status. *See* TIBCO Hawk® Payment Manager Composite, 296
 - service utilization contract for, 22
 - solution architecture documentation, 413–416
 - specification architecture for, 9–10
- Composite service architecture, 20–22
- Composite services
 - in Cached Information Pattern, 304–306
 - in Cascading Control Pattern, 304–305
 - overview of, 303–304
- Composite states, 441–442
- Composite Structure Diagram, UML, 432–434
- Composites
 - implementing TIBCO ActiveMatrix, 307–308
 - information retrieval design patterns, 304–306
 - initiating interaction, 37–39
 - objectives, 293
 - services and applications, 303–304
 - specifying, 294
 - summary review, 308–309
 - understanding, 293–294
- Composites, architecting
 - behavior management, 296–298
 - completing component specifications, 303
 - composite architecture pattern, 295–296
 - mappings, 299–302
 - process of, 303
 - processes, 298–299
- Composition, UML Class Diagram notation, 430–431
- Concepts
 - data model, 216–218
 - naming based on stable, 187
- Concrete definitions, WSDL, 176–177
- Connection factories, JMS destinations, 267–268
- Constraints
 - service specification contents, 127
 - service specification documentation, 422
 - service specification example, 140–141
 - solution architecture documentation, 410
 - technological, for naming standards, 205–206
 - understanding component, 115–116
- Content transformation mediation pattern, 36
- Contention, disk, 333–334
- Context
 - assigning for abstract services, 175
 - service specification contents, 124
 - service specification documentation, 420
 - service specification example, 128–129
 - solution architecture documenting, 410
 - understanding component, 111–112
- Contracts, service usage, 22, 142–144, 404
- Control flow, UML activity diagrams, 437
- Coordination patterns
 - component behavior and, 114–115
 - composite behavior and, 296–298
 - highly available solutions and, 373
 - service specification documenting, 126, 138, 140, 422
 - solution architecture documenting, 414
 - supporting dual, 261–262
 - understanding, 40–44
- Core Pool Size parameter
 - HTTP Connector thread pools, 344
 - JCA thread pool, 342
 - named thread pools, 343
- COTS (commercial off-the shelf) software packages
 - not designed for distributed transactions, 286
 - System-of-Record pattern and, 284–285
 - using Edit-Anywhere-Reconcile-Later pattern, 287–288
- CPU utilization limits, benchmark results, 328–329, 337

Credit interface, Sales Order Service, 99, 101
 Customer Query interface, Sales Order Service, 99–101
 Customer schema, WSDL Sales Order example, 446–447

D

Data access, mainframe interaction intent, 59
 Data center failover, 373
 Data format specifications, solution architecture, 417
 Data management patterns
 Edit-Anywhere-Reconcile-Later, 287–288
 Master-Data-Management, 288–289
 Replicated-Data-with-Transactional-Update, 286–287
 summary review, 290–291
 System-of-Record, 284–285
 System-of-Record-with-Cached-Read-Only-Copies, 285–286
 Data models
 designing common, 224
 designing XML schema, 227–232
 example schema, 235
 organizing schema and interfaces for, 233–234
 representing associations, 226–227
 representing entities, 225–226
 Data, rule services and, 250–252
 Data storage specifications, 417
 Data structures
 common data models, 224–227
 designing deep, 220–222
 designing flattened, 222–223
 designing shallow, 223–224
 designing XML schema, 227–232
 domain models, 215–218
 example schema, 235
 information models, 218–219
 objectives, 215
 organizing schema and interfaces, 233–234
 reusability and, 224
 Data types
 designing XML schema with, 227–228
 entity in data model using, 225–226
 idealized name structures using, 192
 instance names vs. names for, 189
 in WSDL message parts, 176–178
 Database-based intra-site failover pattern, EMS, 383–384
 Decisions, UML activity diagrams, 438
 Dedicated data types, WSDL message parts, 176–178

Deep data structures, 220–222
 Default JMS thread usage, 347–349
 Default node-to-node communications thread usage, 349–351
 Default SOAP/HTTP thread usage, 345–347
 Deferred JMS Acknowledgement Pattern
 ActiveMatrix Service Bus, 390
 ActiveMatrix BusinessWorks, 386
 fault tolerance and high availability, 374
 Deferred payments, case study
 composite mappings of, 299–300
 documenting behavior, 146–149
 Manage Payments process, 75–76
 Payment Manager Composite, 296
 process coordination, 76–77
 process overview, 85, 87–88
 Settle Deferred Payments Service
 Implementation for, 296
 Definitions, adding to WSDL, 160–163
 Delegation coordination pattern, 41–42
 Delegation with confirmation coordination pattern, 41–42
 Deliverables
 hybrid rule-process approaches, 247–248
 Sales Order Service example, 104
 using rule service to assemble process definition, 248–249
 Dependencies. *See also* Observable dependencies and behaviors
 and compatibility, 152
 on packages, 152–153
 as ranges in OSGi versioning, 154–155
 Deployment
 ActiveMatrix, 31–33
 ActiveMatrix Adapters and, 30–31
 ActiveMatrix BusinessWorks, 28–30
 high availability and disaster recovery, 210–211
 service specification contents, 127–128
 service specification documentation, 423
 service specification example, 141–142
 service usage contract, 144
 solution architecture documenting, 416
 Design patterns
 architecture of, 13
 rule service. *See* Rule service, process flow
 Design patterns, used in this book
 basic interaction patterns, 33–34
 coordination patterns, 40–44
 event-driven interaction patterns, 35
 external system access patterns, 37–39
 federations. *See* Service federation
 mediation patterns, 36–37
 service utilization pattern, 17–19

- Destinations
 - asynchronous JMS request-reply, 257–261
 - constraints on destination names, 205–206
 - grouping JMS, 267–268
 - JMS destination names, 183, 201
 - problem with temporary JMS, 257–258
 - SOAP address location indicating, 200
 - subscription interface and, 109–110
 - Development configuration, solution architecture documenting, 416
 - Disk performance limits, benchmark results, 331–334, 337
 - Dispatch Queue, ActiveMatrix BusinessWorks job processing, 365
 - Distributed Federation Pattern, 406
 - Distributed transaction
 - COTS packages not designed for, 286–287
 - with two-phase commit coordination pattern, 43
 - DMZ Component, BusinessConnect, 64
 - Documentation
 - nonfunctional behaviors, 127
 - service-related. *See* Service-related documentation
 - service specification. *See* Service specification documentation
 - solution architecture. *See* Solution architecture, documenting
 - test design. *See* Test design documentation
 - Domain, basing naming on concepts for, 187
 - Domain models
 - information models vs., 218–219
 - Payment Manager, 77–81
 - solution architecture documenting, 410–411
 - understanding data with, 215–218
 - Domain names
 - general-to-specific structured name design, 183–185
 - hierarchical naming authority for, 186
 - notational convention for Internet, 194
 - Dot-separated number sequence, OSGi versioning, 153–154
 - Dual coordination-pattern processing, 261–262
- E**
- Eclipse IDE, Substation ES and, 60
 - Edit-Anywhere- Reconcile-Later pattern, 287–288
 - Editing, avoiding structure mistakes, 434–435
 - EMS (Enterprise Message Service). *See* TIBCO Enterprise Message Service™ (EMS)
 - Engine thread pool
 - ActiveMatrix BusinessWorks Service Engine, 357
 - ActiveMatrix BusinessWorks Service Engine job processing, 362
 - setting ActiveMatrix BusinessWorks Service Engine HTTP server type, 357
 - Enterprise identifiers
 - basing naming on, 187–188
 - including qualifier for functional context, 192–194
 - naming authority complications, 206
 - naming complications in complex organizations, 206–209
 - Enterprise Message Service (EMS). *See* TIBCO Enterprise Message Service™ (EMS)
 - Entities, in data models, 225–226
 - Environment
 - service usage contract specifying, 22
 - solution architecture documenting, 416
 - test harness documenting details of, 318
 - UML execution, 436–437
 - values for address location variables, 209–210
 - WSDL schema location, 197
 - Event-driven interaction, 35, 111
 - Event recognition interaction intent, 59
 - Event Service, Hawk, 55–56
 - Events
 - complex processing of, 425
 - triggering. *See* Triggered behaviors
 - UML activity diagrams for actions, 439
 - Exceptions, raising, 253–257
 - Execution environments, UML, 436–437
 - External system access patterns
 - interaction via adapter, 38–39
 - interaction via non-supported protocol, 39
 - interaction via supported protocol, 37–38
 - overview of, 37
- F**
- Facets, observable dependencies and behaviors, 97
 - Failover. *See* FT and HA (fault tolerance and high availability)
 - Fault tolerance. *See* FT and HA (fault tolerance and high availability)
 - Federation. *See* Service federation
 - Fedex.com, 306
 - File-based intra-site failover pattern
 - client configuration for, 384
 - EMS multi-site with message persistence pattern, 393–396
 - EMS multi-site with no message persistence pattern, 391–393
 - overview of, 382–383

- Files
 - reliable mechanism for locking, 385
 - TIBCO Adapter for Files, 62–63
 - transferring, 56–58
 - WSDL and XSD filenames, 159–160, 195–196
 - FillOrder() operation, case study
 - coordination pattern in, 115
 - overview of, 100–102
 - triggered behaviors, 113
 - Fire-and-forget coordination pattern, 40
 - Flattened data structures, 222–223
 - Flow Limit parameter, ActiveMatrix BusinessWorks, 368
 - Forks, UML activity diagrams, 438
 - Front-end, solution architecture decisions, 240–243
 - FT and HA (fault tolerance and high availability)
 - ActiveMatrix BusinessWorks failover, 385–390
 - common terms, 372–373
 - deferred JMS acknowledgement pattern, 373–374
 - EMS example environment, 391–396
 - EMS failover, 381–385
 - generic site failover, 377–381
 - Intra-Site Cluster Failover Pattern, 374–376
 - objectives, 371–372
 - Service Bus failover, 390–391
 - summary review, 396–399
 - Fulfillment schema, WSDL Sales Order, 449–450
 - Functional context, naming standards. *See also* Naming principles, 192–194, 205–209
 - Functional requirements, service usage contracts, 143–144
 - Funds transfers
 - Payment Manager domain model, 77–81
 - Payment Manager tracking all, 91
 - settlement process, 85, 89
- G**
- General-to-specific structure, of structured name design, 183–185
 - Generalization, UML Class Diagram notation for, 431–432
 - Generic site failover
 - EMS strategy for, 384–385
 - intra-site failover with different host identity pattern, 378–380
 - intra-site failover with same host identity pattern, 380–381
 - overview of, 377
 - storage replication strategies and RPO, 377–378
 - Geographic distribution, naming complications, 207–209
 - GetCustomer() operation, case study, 100–102, 113
 - GetOrder() operation, case study, 104–105
 - GetOrderDetails() operation, case study
 - designing XML schema, 227–232
 - example schema, 227–232
 - organizing schema and interfaces for data model, 233–234
 - WSDL Sales Order Interface example, 443–444
 - Graphs
 - documenting test results, 322–324
 - identifying capacity limit, 327–328
 - identifying CPU utilization limits, 329
 - identifying memory limits, 335
 - identifying network bandwidth limits, 330–331
- H**
- HA (high availability). *See* FT and HA (fault tolerance and high availability)
 - Hawk. *See* TIBCO Hawk®
 - Hawk agents, 49, 55
 - Hawk Display, 51, 54–55
 - Hawk Microagent Adapter (HMA), 53–54
 - Health care claims, 74–75
 - Hierarchical naming authorities, 185–186
 - Hierarchies
 - of architecture, 7–11
 - general-to-specific structured names, 184–185
 - naming complications in complex organizations, 206–209
 - High availability. *See* FT and HA (fault tolerance and high availability)
 - HMA (Hawk Microagent Adapter), 53–54
 - Host identity pattern, intra-site failovers, 378–381
 - HostIdentifier
 - deployment flexibility for HA and DR, 210–211
 - SOAP address location, 201–202
 - values for address location variables, 209–210
 - HTTP
 - interaction via non-supported protocol, 39
 - location naming principles, 201
 - setting ActiveMatrix Service Engine, 357–358
 - thread usage with virtualize policy set and, 352

- HTTP Connector
 - default SOAP/HTTP thread usage, 345–347
 - process starters in ActiveMatrix BusinessWorks using, 359
 - starting jobs from ActiveMatrix Service Bus, 361–363
 - thread pools, 344–345
 - thread usage with threading policy set, 353–356
 - worker thread assignments, 345
- HTTPComponent thread pool, 358
- Hybrid rule-process approaches, 247–248
- I**
- I/O activities, ActiveMatrix BusinessWorks, 362–365, 367–368
- I5/OS, using TIBCO Adapter for IBM i, 61–62
- IBM i platforms
 - TIBCO Adapter for Files, 62–63
 - TIBCO Adapter for IBM i, 61–62
- ICANN (Internet Corporation for Assigned Names and Numbers), 186
- Idealized name structures, 191–192
- Identity pattern, inter-site failover with same-host, 391
- Immediate payments, case study
 - behavior, 136, 139, 146–147
 - Claim Payment Service Implementation, 296
 - composite mappings of, 299, 302
 - coordination, 76, 138–140
 - manage payments process and, 75–76
 - nonfunctional behavior and, 140
 - orchestrating, 298
 - overview of, 83–85
 - Payment Manager architecture pattern, 145–147
 - Payment Manager service interfaces for, 130–132
 - thread usage with threading policy set, 354–356
 - utilization scenario, 129–130
- Implementation
 - TIBCO ActiveMatrix composite, 307–308
 - understanding composite, 294
- Implementation architecture
 - ActiveMatrix Service Bus using Mediation, 27
 - architectural distinctions and, 12
 - services or components, 9, 11
- Implementing SOA: Total Architecture in Practice* (Brown), 425
- In-memory Job pool, ActiveMatrix BusinessWorks, 362
- In-Only architecture pattern
 - orderDelivered() operation, 104
 - orderShipped() operation, 102–103
 - overview of, 33–34
- In-Out architecture pattern, 34
- Incompatible change
 - architecture pattern for deploying, 166–167
 - defined, 152
 - versioning and bug fixes resulting in, 155
 - versioning SOAP interface addresses (endpoints), 168
 - in WSDLs and XML schemas, 163–164
- Inconsistencies, reconciling data, 287–289
- Individual activity benchmarks, 326
- Information models
 - creating XML schema with, 229–232
 - domain models vs., 218–219
- Information retrieval design patterns, 304–306
- Information Storage and Management: Storing, Managing, and Protecting Digital Information* (Somasundaram and Shrivastava), 333
- Input rates
 - determining operating capacity, 316–317
 - documenting test results, 322–324
 - identifying capacity limit, 327–328
 - misleading benchmark results on, 314–316
- Instance names, distinguishing type names from, 189
- Integration and testing, solution architecture documenting, 416–417
- Intended utilization scenarios
 - service specification contents, 125
 - service specification documentation, 420–421
 - service specification example, 129
- Intents, mainframe interaction, 59
- Inter-site failover pattern, ActiveMatrix BusinessWorks, 388–389
- Inter-site failover with different-host identity pattern, 378–379
- Inter-site failover with same-host identity pattern
 - ActiveMatrix Service Bus failover, 391
 - EMS multi-site with message persistence pattern, 393–396
 - EMS multi-site with no message persistence pattern, 391–393
 - generic site failover, 380–381
- Interaction patterns
 - basic, 33–34
 - event-driven, 35
 - external system access, 37–39

- Interface definitions
 - service specification contents, 125
 - service specification documentation, 421
 - service specification example, 129–132
 - WSDL, 176–178
- Interface major version numbers, structured name design, 191
- Interfaces
 - dependencies and compatibility of, 152
 - distinct WSDL namespace URI for, 190
 - Payment Manager, 82–83
 - solution architecture decisions, 240–243
 - solution architecture documenting, 414
 - UML Class Diagram notation for, 428–429
- Internal engine architecture, ActiveMatrix BusinessWorks, 358–359
- Internet Corporation for Assigned Names and Numbers (ICANN), 186
- Internet naming system
 - general-to-specific structure of, 183–185
 - hierarchical naming authorities in, 185–186
 - notational convention for domain names, 194
 - technology constraints on, 205
- Internet Server, TIBCO Managed File Transfer, 56
- Intra-Site Cluster Failover Pattern, 374–376
- Intra-site failover pattern
 - ActiveMatrix BusinessWorks built-in, 387–388, 390–391
 - ActiveMatrix BusinessWorks cluster, 388–389
 - ActiveMatrix Service Bus cluster, 391
 - with different host identity pattern, 378–380
 - EMS strategy for, 382–384
 - recovery point objectives, 373
 - with same host identity pattern, 380–381
- Invoke Partner thread pool, 358, 365–366
- IOPS (I/O operations per second), disk performance, 331–334
- IP redirectors, load distribution, 266
- ISeries, 58–63
- J**
 - JCA thread pool, 342–343
 - JDBC, external access interaction via, 39
 - JMS Binding thread pool
 - ActiveMatrix nodes, 343
 - default JMS thread usage, 347–349
 - default node-to-node communications thread usage, 349–351
 - thread usage with threading policy set, 355–356
 - thread usage with virtualize policy set and initiation of, 352–353
 - worker thread assignments, 345
 - JMS (Java Message Service)
 - ActiveMatrix BusinessWorks Service Engine thread usage, 361–364
 - asynchronous request-reply interactions, 257–261
 - Deferred JMS Acknowledgement Pattern, 373–374
 - load distribution, partitioning between servers, 267–269
 - load distribution, two-tier, 279–280
 - load distribution, with queues, 266
 - name requirements, 183
 - naming constraints, 205
 - naming principles, 201
 - JMS process starters, 360–361
 - JMSCorrelationID property, asynchronous request-reply, 258–261
 - JMSReplyTo property, asynchronous request-reply, 258–261
 - JNDI lookups, partitioning JMS message load, 268
 - Jobs, ActiveMatrix BusinessWorks
 - Activation Limit parameter, 367–368
 - Flow Limit parameter, 368
 - Max Jobs parameter, 365–367
 - processing, 362–366
 - starting from Service Bus, 361–364
 - Step Count parameter, 368–369
 - Joins, UML activity diagrams, 438
- K**
 - Keep Alive Time parameter, thread pools, 343–344
- L**
 - Licenses, message service, 27
 - Lines of business
 - naming authority complications, 206
 - naming complications in complex organizations, 206–209
 - naming principles for WSDL and XSD filenames, 196
 - naming principles in functional context for, 193–194
 - Load distribution
 - EMS client connections, 269–271
 - IP redirectors for, 266
 - JMS queues for, 266
 - objectives, 265–266
 - partitioning JMS message load between servers, 267–269
 - patterns preserving partial ordering, 278–280
 - patterns preserving total sequencing, 275–278
 - sequencing problem, 273–275

- Load distribution, *continued*
 - with Service Bus, 271–273
 - summary review, 280–281
- Logging, Hawk Event Service, 55–56
- Lookup and cross reference, information retrieval, 306

M

- MagicDraw tool, 434
- Mainframe-related products, and iSeries interaction, 58–63
- Major version numbers
 - namespace names and, 159–160
 - OSGi versioning and, 153–154
 - rules for WSDLs and schemas, 164, 191
- Manage payments, case study
 - Payment Manager specification, 74
 - Payment Manager specification, domain model, 77–81
 - process coordination, 76–77
 - processes of, 75–76
- Managed File Transfer product portfolio, TIBCO®, 56–58
- Manufacturer schema, 447–448
- Mapping processes. *See* Process mapping
- Master-Data-Management pattern, 288–289
- Max_connections parameter, EMS client load distribution, 270–271
- Max Jobs tuning parameter, ActiveMatrix BusinessWorks, 365–367
- Max Pool Size parameter
 - default SOAP/HTTP thread usage, 347
 - JCA thread pool, 342
 - named thread pools, 344
- Maximum size parameter, thread pools, 343–344
- Mediation patterns
 - ActiveMatrix BusinessWorks, 30
 - ActiveMatrix Service Bus, 27–28
 - ActiveMatrix Service Grid, 29
 - overview of, 36–37
- Memory limits
 - interpreting benchmark results, 335–336
 - setting Max Jobs parameter in ActiveMatrix BusinessWorks, 367–368
 - using benchmark results on, 337
- Message format specifications, solution architecture, 417
- Message service. *See* TIBCO Enterprise Message Service™ (EMS)
- Messages
 - avoiding deep data structure for, 222
 - avoiding flattened data structure for, 223
 - using shallow data structures for, 223–224
- Micro version numbers, OSGi, 154

- Microagent interfaces, Hawk, 54–55
- Migration, solution architecture documenting, 416
- Milestone-level status, service interfaces, 107–108
- Minimum size parameter, JMS Binding thread pool, 343
- Minor version numbers
 - OSGi, 154
 - placement for WSDLs and XML schemas, 159–160
 - rules for versioning WSDLs and XML schemas, 164
 - structured names incorporating interface, 191
- Multi-word fields, name structure with, 189–190

N

- Named thread pools, ActiveMatrix nodes, 343–344
- Namespace URIs, WSDL and schemas, 159–160
- Namespaces, responsibility for subordinate, 186
- Naming authorities
 - complications of, 206
 - hierarchical, 185–186
- Naming principles
 - functional context, 192–194
 - idealized name structures, 191–192
 - operation names, 200
 - port type, service, and binding names, 198–200
 - schema locations, 197
 - schema shared data types, 204–205
 - schema types specific to interface and operation, 203–204
 - SOAP action names, 203
 - SOAP address location, 200–202
 - WSDL and XSD filenames, 195–196
 - WSDL and XSD namespace URIs, 194–195
 - WSDL names, 196–197
 - WSDL-specific schema location, 197–198
 - WSDLmessage names, 197–198
- Naming standards
 - abstract services concept, 174–175
 - avoiding acronyms and abbreviations, 188
 - basing on business processes, 187
 - basing on domain concepts, 187
 - basing on enterprise identifiers, 187–188
 - in complex organizational structures, 206–209
 - defining routing strategies, 181–182
 - defining search strategies, 181

- deployment flexibility for HA and DR, 210–211
- developing your own, 211–212
- difficulty of changing names, 180
- distinct WSDL namespace URI for each interface, 190
- distinguishing types from instances, 189
- general-to-specific structure, 183–185
- hierarchical naming authorities, 185–186
- importance of, 180
- interface major version numbers, 191
- multi-word fields, 189–190
- naming authorities, 206
- objectives, 173–174
- overview of, 174
- relating abstract and WSDL-defined services, 178–180
- summary review, 212–213
- technology constraints on, 205–206
- things requiring, 182–183
- using this chapter, 174
- values for address location variables, 209–210
- WSDL interface definitions, 176–178
- Nested retrieval, 111
- Network bandwidth limits, benchmark results, 329–331, 337
- Network interface cards (NICs), network bandwidth limits, 331
- Nonfunctional behavior
 - characterizing observable behavior, 116–117
 - service specification contents, 127
 - service specification documentation, 422–423
 - service specification example, 140–141
 - solution architecture documentation, 414–415
- Nonfunctional requirements
 - service usage contract specifying, 144
 - solution architecture documentation, 412–413
- Notification Router, Payment Manager Composite, 296
- Notifications
 - Hawk Event Service, 55
 - service specification documenting, 126
- Nouveau Health Care, case study
 - objectives, 67–68
 - Payment Manager service specification, 73
 - Payment Manager specification, domain model, 77–81
 - Payment Manager specification, interfaces, 82–83

- Payment Manager specification, process overview, 74–77
- Payment Manager specification, processes, 83–90
- service-related documentation. *See* Service-related documentation
- solution architecture, 68–73
- summary review, 91
- Number scheme, OSGi versioning, 153–154

O

- Object-Oriented Modeling and Design with UML, Second Edition* (Blaha and Rumbaugh), 218
- Observable architecture, 414
- Observable behavior, defined, 96
- Observable dependencies and behaviors
 - black box perspective, 96–97
 - composites not suitable for black box characterization, 117–118
 - constraints, 115–116
 - context, 111–112
 - context summarizing, 124
 - coordination, 114–115
 - facets of, 97
 - nonfunctional behavior, 116–117
 - objectives, 95–96
 - observable state, 114
 - summary review, 119
 - triggered behaviors, 113–114
 - usage scenarios, 112–113
- Observable dependencies and behaviors, Sales Order Service example
 - avoiding caches with nested retrieval, 111
 - observable state and cached information, 108–111
 - observable state information, 104–108
 - order delivered, 104
 - order shipped, 102–103
 - placing order, 98–102
- Observable dependencies, defined, 96
- Observable state
 - cached information and, 108–111
 - information, 104–108
 - observable behavior characterized by, 114
 - orderShipped() operation, 102–103
 - service specification contents, 125–126
 - service specification documentation, 421
 - service specification example, 134–135
 - solution architecture documentation, 414
- ObtainPayment() operation, case study, 99, 101–102, 113
- ODS (Operational Data Store), 295–296
- One-line description
 - defined, 122
 - service specification content, 124

- One-line description, *continued*
 - service specification documentation, 419
 - service specification example, 128
 - One-way fire-and-forget messages, 268
 - Open Services Gateway initiative (OSGi)
 - Framework
 - dependencies on packages, 152–153
 - overview of, 151
 - version number scheme, 153–154
 - Operating capacity
 - determining, 316–317
 - interpreting benchmark results, 327–328
 - misleading benchmark results, 315
 - setting Max Jobs parameter in ActiveMatrix BusinessWorks, 366
 - Operation invocations, service specification
 - documenting, 126
 - Operation names, WSDL, 200
 - Operational Data Store (ODS), 295–296
 - Operational information, service interfaces, 106–107
 - Optional field, XML schema, 162–163
 - Orchestration design pattern, 297–298
 - Order Fulfillment interface, Sales Order Service
 - coordination patterns, 115–116
 - orderDelivered() operation, 104
 - orderShipped() operation, 102–103
 - overview of, 100–101
 - Order status, deep data structure for, 220–222
 - Order-to-Delivery business process. *See* Sales Order Service example
 - OrderDelivered() operation, case study, 104–105, 108, 115
 - OrderShipped() operation, case study, 102–103, 108, 115
 - Organizations
 - establishing naming standard for, 211–212
 - naming complications in complex, 205–209
 - service usage contract specifying responsible, 142–143
 - Orthogonal states, 441–442
 - OSGi (Open Services Gateway initiative)
 - Framework
 - dependencies on packages, 152–153
 - overview of, 151
 - version number scheme, 153–154
 - Out-In architecture pattern, 34
 - Out-Only architecture pattern, 34
 - Output rates
 - determining operating capacity, 316–317
 - documenting test results, 322–324
 - identifying capacity limit, 327–328
 - identifying CPU utilization limits, 328–329
 - misleading benchmark results on, 314–316
 - Over-partitioning, JMS message load
 - between servers, 269
 - Overhead benchmarks, for complex components, 325
 - Overruling, advanced Hawk rules, 50–52
- ## P
- Packages, version dependencies and, 152–153
 - Parameters
 - ActiveMatrix BusinessWorks tuning, 365–369
 - default SOAP/HTTP thread usage, 346–347
 - documenting test results, 322–324
 - documenting testing of experimental, 321–322
 - individual activity benchmark, 326
 - JMS process starters in ActiveMatrix BusinessWorks, 360–361
 - overhead benchmark, 325
 - thread pools in ActiveMatrix nodes, 342–345
 - Tomcat thread pool, 357
 - Parenthetical () brackets, OSGi versioning, 155
 - Part-whole relationships, UML Class Diagram notation for, 430–431
 - Partial order sequencing
 - load distribution patterns preserving, 278–280
 - overview of, 274–275
 - Partitioning, JMS message load between servers, 267–269
 - Payer accounts, case study, 77–78
 - Payment domain model, case study, 80–81
 - Payment Manager composite
 - behavior management, 296–298
 - composite processes, 298–299
 - implementation, 307–308
 - overview of, 295–296
 - Payment Manager specification, case study
 - domain model, 77–81
 - interfaces, 82–83
 - process overview, 74–77
 - processes, 83–90
 - service-related documentation. *See* Service-related documentation
 - service specification example. *See* Service specification
 - usage context, 73
 - Performance
 - benchmarking. *See* Benchmarking
 - service usage contract specifying, 144

- solution architecture documenting, 412, 417
 - storage replication strategies and, 377–378
 - System-of-Record pattern limitations, 285
 - tuning. *See* Tuning
 - typical curve showing, 314–315
 - Performance targets, high availability, 373
 - Persisted message read times, EMS failover strategy, 385
 - PERSISTENT mode, Deferred JMS Acknowledgement Pattern, 374
 - Phone schema, WSDL Sales Order, 448
 - PlaceOrder() operation, case study
 - avoiding caches with nested retrieval, 111
 - coordination pattern, 115
 - overview of, 98–102
 - Platform Server, TIBCO Managed File Transfer product portfolio, 56
 - Policy
 - hierarchical naming authorities managing, 186
 - intents, 45
 - Ports
 - name structure defining routing strategies, 180–181
 - UML Class Diagram notation for, 436
 - WSDL type names for, 198–200
 - Priority parameter, named thread pools, 344
 - Process claim, case study
 - accounts and fund transfers, 77–81
 - basic concepts, 74–75
 - process coordination, 76–77
 - processing claims from providers, 74
 - Process coordinator, 44–45
 - Process manager
 - rule service as, 246–247
 - rule service assembling process definition, 248–249
 - rule service directing, 249–250
 - rule service separated from, 245–246
 - Process mapping
 - architecture pattern for incompatible change, 166–167
 - composites, 21–22, 298–299, 303
 - deferred payment case study, 88
 - immediate payment case study, 83–86
 - service utilization, 19
 - settlement account case study, 90
 - simple test, 318–321
 - solution architecture, 14–16
 - Process models
 - as architecture view, 4–6
 - in hierarchy of architecture, 7–11
 - service utilization, 18–19
 - Process-pattern mapping
 - as architecture view, 6–7
 - in hierarchy of architecture, 7–11
 - showing usage scenarios with, 98, 113
 - solution architecture documenting, 411
 - Process starters
 - ActiveMatrix BusinessWorks internal engine architecture, 358–359
 - JMS, 360–361
 - Processes
 - Payment Manager, 83–90
 - Payment Manager overview of, 74–77
 - Product portfolio. *See* TIBCO ActiveMatrix® Product Portfolio
 - Product Query interface, 99–100
 - Product schema, WSDL Sales Order, 448–449
 - ProductChange() operation, Sales Order Service, 110–111
 - Program-to-program interaction intent, mainframes, 59
 - Promoted services, Service Bus load distribution, 272–273
 - Protocols
 - BusinessConnect facilitating B2B, 63–64
 - external access interaction via non-supported, 39
 - external access interaction via proprietary, 38
 - external access interaction via supported, 37–38
 - Provider account, case study, 77–78
 - Provider Settlement Record, 83–86
- ## Q
- Query operations, Microagent, 54
 - Queue message delivery semantic, 35
 - Queue Pattern, EMS client load distribution, 270–271, 393–396
 - Queues, JMS load distribution, 266
- ## R
- RAID arrays
 - identifying disk performance limits, 333
 - preventing data lost in data center, 377
 - Ranges, dependencies in OSGi versioning as, 154–155
 - Read requests, disk performance limits, 331–334
 - Recovery point objective (RPO), 372–374, 377–378
 - Recovery time objective (RTO), 372–373, 390
 - Recursive structure, UML notation, 433
 - Refactored architecture, 242–243
 - Reference architecture
 - documenting design pattern, 13
 - service utilization pattern as, 17–19, 24
 - Reference (or master) data, managing, 64–65

- Referenced components, service specification content, 421
- Referenced interfaces
 - applying Threading Policy Sets to, 354–356
 - applying Virtualize Policy Set to, 351–352
 - deploying, 32, 36
 - service specification content, 125
 - service specification documentation, 423
 - service specification example, 132–134
- Rejection Policy parameter, named thread pools, 344
- Relationships, data model, 216–218
- Remote Domain Pattern, federation, 405–406
- Rendezvous. *See* TIBCO Rendezvous®
- Replicated-Data-with-Transactional-Update pattern, 286–287
- Repository issues, federation, 403
- Representations
 - representing in data models, 224–227
 - XML schema design using, 228–229
- Request-reply coordination pattern
 - Deferred JMS Acknowledgement Pattern, 373–374
 - overview of, 40–41
 - partitioning JMS message load, 268
- Request-reply test scenario, documenting test harness, 317–321
- Results, benchmarking test
 - documenting, 322–324
 - identifying capacity limit, 327–328
 - identifying CPU utilization limits, 328–329
 - identifying disk performance limits, 331–334
 - identifying memory limits, 335–336
 - identifying network bandwidth limits, 329–331
 - identifying test harness limits, 336
 - misleading, 314–316
 - using, 326–327
- Reusability
 - of data types in XML schema, 227–228
 - designing data structures and, 224
- Risks, solution architecture documentation, 410
- Rocket Stream® Accelerator, 56–57
- Routing mediation pattern, 36–37
- Routing strategies, 180
- RPO (recovery point objective), 372–374, 377–378
- RTO (recovery time objective), 372–373, 390
- Rule service, process flow
 - acquiring data, 250–252
 - assembling process definition, 248–249
 - directing process manager, 249–250
 - hybrid rule-process approaches, 247–248
 - overview of, 244–245
 - as process manager, 246–247
 - rule client providing data, 250–251
 - separating from process manager, 245–246
- Rulebase Editor, Hawk rules, 49–53
- Rules
 - Hawk, 49–55
 - versioning WSDLs and XML schemas, 164
- S**
- Sales Order Interface schema, WSDL, 450–451
- Sales Order Service example
 - applying constraints, 115–116
 - avoiding caches with nested retrieval, 111
 - capturing coordination, 115–116
 - documenting nonfunctional behavior, 116–117
 - observable state and cached information, 108–111
 - observable state information, 104–108
 - order delivered, 104
 - order shipped, 102–103
 - overview of, 97–98
 - placing order, 98–102
 - Sales Order Interface schema, 450–451
 - Sales Order Interface WSDL, 443–444
 - Sales Order schema, 451–452
 - Sales Order schema, WSDL, 451–452
 - showing dependencies, 112
 - usage scenarios, 112–113
 - WSDLs and schemas. *See* WSDL (Web Services Description Language) and Schemas
- Sales Order Status Interface
 - coordination patterns, 115
 - milestone-level status, 108
 - orderDelivered() operation, 104–105
 - orderShipped() operation, 102–103
 - showing dependencies, 112
- SCA (Service Component Architecture)
 - design, 31–33, 183
- Scalability, solution architecture documenting, 412
- Scenario benchmarks, complex components, 326–327
- Scheduling, with Hawk rules, 51–52
- Schema shared data types, naming principles, 204–205
- Schema types, naming principles, 203–204
- Schemas
 - WSDL. *See* WSDL (Web Services Description Language) and schemas
 - XML. *See* XML schemas
 - XSD. *See* XSD (XML Schema Definition)

- Scope
 - data model schema and interfaces, 233–234
 - information model, 218–219
- Search strategies, defined by name structure, 180
- Secondary server startup time, EMS failover strategy, 384–385
- Security
 - solution architecture documenting, 412–413
 - specifying in service usage contract, 144
- Selector property, JMS request-reply interactions, 258–261
- Sequence manager pattern, 276–278
- Sequencing
 - load distribution and problem of, 273–275
 - patterns preserving partial order, 278–280
 - patterns preserving total, 275–278
- Sequencing key, 279–280
- Servers
 - EMS, 26
 - EMS failover strategy, 384–385
- Service abstract, service specification documenting, 420
- Service architecture
 - building-block design patterns. *See* Building-block design patterns
 - component life cycle and, 22–23
 - composite, 20–22
 - documenting, 144–149
 - overview of, 17
 - reasons for architectural distinctions, 12
 - service implementation architecture, 9, 11
 - service specification architecture, 9–10
- Service Component Architecture (SCA)
 - design, 31–33, 183
- Service consumers
 - component life cycle governance of, 22–23
 - service usage contract specifying, 143
- Service context, service specification documenting, 420
- Service domains, standardizing technology, 407
- Service Engine. *See* TIBCO ActiveMatrix BusinessWorks™ Service Engine
- Service federation
 - access control issues, 402–403
 - basic federation pattern, 403–405
 - concept of, 402
 - Distributed Federation Pattern, 406
 - objectives, 401
 - with Remote Domain Pattern, 405–406
 - repository issues, 403
 - standardizing service domain technology, 407
 - summary review, 407–408
- Service interface specifications, documenting, 417, 423
- Service names, WSDL, 198–200
- Service one-line description, service specification documenting, 419
- Service providers, component life cycle governance of, 22–23
- Service-related documentation
 - objectives, 121–122
 - service abstract, 122–123
 - service architecture, 144–149
 - service one-line description, 122
 - service specification. *See* Service specification
 - service usage contracts, 142–144
 - summary review, 149–150
- Service specification
 - abstract, 124, 128
 - constraints, 127, 140–141
 - context, 124, 128–129
 - coordination, 126, 138, 140
 - deployment specifics, 127–128, 141–142
 - intended utilization scenarios, 125, 129
 - interface definitions, 125, 129–132
 - nonfunctional behavior, 127, 140–141
 - observable state, 125–126, 134–135
 - one-line description, 122, 124, 128
 - referenced interfaces, 125, 132–134
 - service abstract, 128
 - service context, 128–129
 - triggered behaviors, 126, 136–139
- Service specification documentation
 - abstract, 420
 - appendices, 423
 - constraints, 422
 - context, 420
 - coordination, 422
 - deployment, 423
 - intended utilization scenarios, 420–421
 - interface definitions, 421
 - nonfunctional behavior, 422–423
 - observable state, 421
 - one-line description, 419
 - overview of, 124
 - referenced interfaces, 421
 - triggered behaviors, 422
- Service usage contracts
 - basic federation pattern, 404
 - elements of, 142–144
 - overview of, 22
- Service utilization pattern, 17–19
- Services
 - building wrappers around back-end systems, 243–244
 - composite, 303–304

- Services, *continued*
 - defined, 17
 - implementing as composites, 20
 - monitoring status of. *See* TIBCO Hawk®
 - returning variant business responses, 252–257
 - solution architecture documentation, 413–416
- Settle Deferred Payments, case study
 - behavior, 136, 138, 145–146, 149
 - composite mappings, 301
 - composite processes, 296
 - Manage Payments process, 75–76
 - process coordination, 76–77
 - process mapping, 299, 301
 - Service Implementation, 296
 - utilization scenario, 129–130
- Settlement Accounts, case study
 - concepts, 79–80
 - defined, 77–78
 - immediate payment process overview, 83–86
 - overview of, 78–79
 - process overview, 85, 89–90
- Settlement Service Implementation, 296
- Shallow data structures, 223–224
- Shipping of goods, example, 106–107
- Simple type values, replacing with identical
 - simple type in XML, 161–162
- Single-threaded design pattern, 275–276
- Site disaster recovery. *See also* FT and HA (fault tolerance and high availability)
 - service usage contract specifying, 144
 - solution architecture documentation, 412
- Site failover patterns. *See* Generic site failover
- SOAP action names
 - major version numbers in, 191
 - principles for, 203
 - version numbers in, 160
 - versioning, 168–169
 - WSDL interface definitions and, 176
- SOAP address location, naming principles
 - for, 200–202
- SOAP/HTTP call, thread usage, 345–349
- SOAP interface addresses (endpoints)
 - major version numbers in names, 191
 - schema versioning rules, 164
 - versioning, 168–170
 - WSDL interface definitions and, 176–178
- Solution architecture
 - case study, 68–73
 - in hierarchy of architecture, 8
 - overview of, 14–16
 - reasons for architectural distinctions, 11–12
 - service or component implementation
 - architecture, 9, 11
 - service or component specification
 - architecture, 9–10
 - views characterizing, 24
- Solution architecture decisions
 - asynchronous JMS request-reply
 - interactions, 257–261
 - business exceptions, 252–257
 - rule service governing process flow, 244–250
 - rule services and data, 250–252
 - separating interface and business logic, 240–243
 - summary review, 262–264
 - supporting dual coordination patterns, 261–262
 - using services for back-end systems, 243–244
- Solution architecture, documenting
 - addressing nonfunctional solution
 - requirements, 412–413
 - appendices, 417
 - architecture pattern, 411
 - business objectives and constraints, 409–410
 - business process inventory, 410
 - business processes, 411–412
 - components and services, 413–416
 - deployment, 416
 - domain model, 410–411
 - integration and testing requirements, 416–417
 - solution context, 410
- Specification architecture
 - reasons for architectural distinctions, 11–12
 - service or component, 9–10, 17
- Specification, composite, 294
- Sponsored top-level domains, 186
- Square brackets [], ranges in OSGi versioning, 155
- SRDF (Symmetrix Remote Data Facility),
 - storage replication, 377
- Standardized service domain technology,
 - service federation, 407
- State machines, UML notation reference, 441–442
- Step Count parameter, ActiveMatrix
 - BusinessWorks, 368–369
- Storage replication strategies, 377–378
- Straight-through mediation pattern, 36
- Structure, UML Class Diagram notation
 - avoiding common editing mistakes, 434–435
 - Composite Structure Diagram, 432–434
 - execution environment, 435–436
 - swimlanes with, 440
 - UML Class Diagram notation for, 432

- Structured activity, UML activity diagrams, 438
 - Structured name design principles
 - avoiding acronyms and abbreviations, 188
 - basing on business processes, 187
 - basing on domain concepts, 187
 - basing on enterprise identifiers, 187–188
 - distinct WSDL namespace URI for each interface, 190
 - distinguishing types from instances, 189
 - general-to-specific structure, 183–185
 - hierarchical naming authorities, 185–186
 - interface major version numbers, 191
 - planning for multi-word fields, 189–190
 - Subdomains, in structured names, 183–185
 - Subscriptions
 - Microagent, 55
 - Sales Order Service, 109–110
 - Subtrees, in structured names, 183–185
 - Swimlanes, in UML activity diagrams, 439–440
 - Symmetrix Remote Data Facility (SRDF), storage replication, 377
 - Synchronous coordination patterns, 261–262
 - Synchronous request-reply coordination pattern, 40–41, 268
 - Synchronous storage replication strategies, generic site failover, 377–378
 - System-of-Record pattern, 108, 284–285
 - System-of-Record-with-Cached-Read-Only-Copies pattern, 285–286
- T**
- Technology constraints, naming standards, 205–206
 - Temporary destinations, for JMS replies, 257–258
 - Test design documentation
 - experimental parameters, 321–322
 - solution architecture documentation, 416–417
 - test harness architecture pattern, 317–318
 - test harness process mapping, 318–321
 - test results, 322–324
 - Test harnesses
 - documenting architecture pattern, 317–319
 - documenting parameters, 321
 - documenting process mapping, 318–321
 - interpreting benchmark results, 336
 - misleading results of, 316
 - Testing Hawk rules, 50–51
 - Thread pools, 342–345, 357–358
 - Thread usage, ActiveMatrix Service Bus nodes
 - default JMS, 347–349
 - default node-to-node communications, 349–351
 - default SOAP/HTTP, 345–347
 - with threading policy set, 353–356
 - with virtualize policy set, 351–353
 - Threading policy set, 353–356
 - TIBCO ActiveMatrix BusinessWorks™
 - architecture fundamentals of, 27–28
 - failover, 385–390
 - plug-in for Data Conversion, 60–61
 - preserving partial ordering in, 278–280
 - single-threaded solutions with, 275–276
 - TIBCO ActiveMatrix BusinessWorks™ Service Engine
 - ActiveMatrix BusinessWorks internal engine architecture, 358–359
 - ActiveMatrix BusinessWorks jobs processing, 362–365
 - ActiveMatrix BusinessWorks tuning parameters, 365–369
 - JMS process starters, 360–361
 - starting jobs from Service Bus, 361–362
 - summary review, 369–370
 - thread pools, 357–358
 - TIBCO ActiveMatrix® Adapters
 - architecture fundamentals of, 30–31
 - external system interaction via, 38–39
 - single-threaded solutions with, 276
 - TIBCO ActiveMatrix® Administrator
 - ActiveMatrix Service Bus providing, 27
 - managing ActiveMatrix BusinessWorks processes, 29
 - managing ActiveMatrix Service Bus load distribution, 272
 - TIBCO Administrator vs., 26
 - TIBCO ActiveMatrix® composite implementation, 307–308
 - TIBCO ActiveMatrix® Decisions, 246
 - TIBCO ActiveMatrix® failover, 390–391
 - TIBCO ActiveMatrix® Lifecycle Governance Framework, 403, 407
 - TIBCO ActiveMatrix® Product Portfolio
 - ActiveMatrix BusinessWorks, 27–28
 - ActiveMatrix Adapters, 27–28
 - ActiveMatrix Service Bus, 27–28
 - ActiveMatrix Service Grid, 27–28
 - overview of, 27
 - TIBCO ActiveMatrix® Service Bus
 - ActiveMatrix BusinessWorks process deployment with, 28–29
 - ActiveMatrix Service Grid including all components for, 28–29
 - architecture fundamentals of, 27–28
 - implementing two-tier load distribution preserving partial ordering, 279–280
 - initiating interaction via supported protocol, 37–38
 - interaction design patterns used by, 33–34

- TIBCO ActiveMatrix® Service Bus, *continued*
 - load distribution in, 271–273
 - mediation implementation type, 36–37
 - policies, 44–45
 - service federation access control with, 403
 - starting ActiveMatrix BusinessWorks jobs from, 361–364
- TIBCO ActiveMatrix® Service Bus nodes
 - default JMS thread usage, 347–349
 - default node-to-node communications thread usage, 349–351
 - default SOAP/HTTP thread usage, 345–347
 - HTTP Connector thread pools, 344–345
 - JCA thread pool, 342–343
 - JMS Binding thread pool, 343
 - named thread pools, 343–344
 - overview of, 341–342
 - summary review, 369–370
 - thread usage with threading policy set, 353–356
 - thread usage with virtualize policy set, 351–353
 - virtualization thread pool, 343
 - worker thread assignments, 345
- TIBCO ActiveMatrix® Service Grid, 27–29, 62
- TIBCO ActiveSpaces®, 284, 306
- TIBCO® Adapter
 - external system interaction via, 38–39
 - for Files, 62
 - for IBM i deployment, 59, 61–62
- TIBCO Administrator™
 - “Classic” process management with, 29
 - EMS servers, 26
 - TIBCO ActiveMatrix® Adapters, 30–31
- TIBCO architecture fundamentals
 - ActiveMatrix BusinessWorks, 28–30
 - ActiveMatrix Adapters, 30–31
 - ActiveMatrix deployment options, 31–33
 - ActiveMatrix Product Portfolio, 27
 - ActiveMatrix Service Bus, 27–28
 - ActiveMatrix Service Bus policies, 44–45
 - ActiveMatrix Service Grid, 28–29
 - design patterns, 33–44
 - Enterprise Message Service (EMS) product, 26
 - overview of, 25
 - summary review, 46
- TIBCO® *Architecture Fundamentals* book, 25
- TIBCO Business Studio™
 - composite implementation, 307
 - configuring ActiveMatrix Adapters, 30
 - TIBCO Designer Add-in for, 28
- TIBCO BusinessConnect™, 63–64, 66
- TIBCO BusinessEvents®
 - combining process manager/rule service roles, 247
 - rule service role of, 246
 - usage of, 252
 - using System-of-Record data management pattern, 284
- TIBCO BusinessWorks™ SmartMapper, 306
- TIBCO “Classic,” 29
- TIBCO Collaborative Information Manager™, 64–65, 289
- TIBCO Designer™ Add-in TIBCO Business Studio™, 28–31
- TIBCO EMS Central Administration, 26
- TIBCO Enterprise Message Service™ (EMS)
 - architecture fundamentals of, 26
 - availability environment example, 391–396
 - client connection load distribution, 269–271
 - distributing load with JMS queues, 266
 - event-driven interaction patterns used by, 35
 - failover, 381–385
 - Hawk Agent interaction with, 56
 - installing with ActiveMatrix Service Bus, 27
 - recovery point objectives for, 374
 - TIBCO Adapter for IBM i using, 61
 - TIBCO BusinessConnect using, 64
 - TIBCO Substation ES deployment using, 60
 - TIBCO Adapter for Files using, 62–63
- TIBCO® *Enterprise Message Service™ User’s Guide*, 270
- TIBCO Hawk®
 - adapters, 56
 - creating rules, 49–53
 - Event Service, 55–56
 - Hawk Agent, 49
 - Hawk Display, 55
 - Hawk Microagent Adapter (HMA), 53–54
 - Microagent interfaces, 54–55
 - overview of, 48–49
- TIBCO® Managed File Transfer product
 - portfolio, 56–58
- TIBCO Mainframe Service Tracker™, 60–61
- TIBCO products
 - mainframe and iSeries integration, 58–63
 - objectives, 47–48
 - summary review, 65–66
 - TIBCO BusinessConnect, 63–64
 - TIBCO Collaborative Information Manager, 64–65
 - TIBCO Hawk. *See* TIBCO Hawk®
 - TIBCO Managed File Transfer, 56–58

- TIBCO Rendezvous®
 - Hawk Agent interaction with, 49, 56
 - TIBCO Adapter for IBM i using, 61
 - TIBCO BusinessConnect using, 64
 - TIBCO Substation ES deployment using, 60
 - TIBCO Adapter for Files using, 62–63
 - TIBCO Substation ES™, 60
 - Time-based events, service specification, 126
 - Tomcat thread pool, 357
 - Top-level domain names, naming authorities, 186
 - Topic message delivery semantic, event-driven interaction pattern, 35
 - Topic pattern, EMS client load distribution, 269–271
 - Total sequencing, load distribution patterns preserving, 275–278
 - Traffic, partitioning JMS message load, 268
 - Transactional Acknowledgement Mode, JMS process starters, 360–361
 - Transfer rates, disk performance, 332
 - Transitions, states and, 441
 - Trees, general-to-specific structured name design, 183–185
 - Triggered behaviors
 - capturing coordination when modeling, 115
 - characterizing observable behavior, 113–114
 - composite processes executing, 298–299
 - documenting service architecture, 145–149
 - mapping onto components of composites, 299–300
 - orderDelivered() operation, 105
 - orderShipped() operation, 102–103
 - placeOrder() operation, 100, 102
 - service specification contents, 126
 - service specification documentation, 422
 - service specification example, 136–139
 - Trivial processes, overhead benchmarks with, 325
 - Try ... catch constructs, business variations, 253, 256–257
 - Tuning
 - ActiveMatrix Service Bus nodes. *See* TIBCO ActiveMatrix® Service Bus nodes
 - objectives, 341
 - Service Engine. *See* TIBCO ActiveMatrix BusinessWorks™ Service Engine
 - summary review, 369–370
 - Two-threaded test harness mapping, 321
 - Two-tier load distribution pattern, 279–280
- U**
- UML Class Diagram notation
 - associations, 429–430
 - classes, 427–428
 - common structure editing mistakes, 434–435
 - composite structure diagrams, 432–434
 - execution environments, 436–437
 - generalization, 431–432
 - interfaces, 428–429
 - part-whole relationship, 430–431
 - ports, 436
 - structure, 432
 - UML notation reference
 - activity diagrams, 437–440
 - Class Diagram. *See* UML Class Diagram notation
 - collaborations, 440
 - state machines, 441–442
 - Unified Modeling Language Reference Manual* (Rumbaugh, Jacobsen, and Booch), 426
 - Unique names
 - basing on stable concepts, 187
 - establishing naming standard, 211–212
 - with hierarchical naming authorities, 185–186
 - importance of, 180
 - for WSDL names, 196
 - UNIX, TIBCO Adapter for IBM i, 61
 - Updates
 - Edit-Anywhere-Reconcile-Later pattern and, 287
 - Microagent operations, 54
 - URIs (uniform resource identifier)
 - general-to-specific structured name design, 184–185
 - name structure defining routing strategies, 180–181
 - naming principles for WSDL and XSD, 194–196
 - SOAP location names, 200–202
 - uniqueness of WSDL name, 196
 - WSDL interfaces, 190
 - WSDL message names, 197–198
 - URLs (uniform resource locators)
 - general-to-specific structured name design, 184–185
 - naming principles for WSDL and XSD, 195–196
 - uniqueness of WSDL name, 196
 - Usage scenarios
 - applying constraints, 115–116
 - characterizing observable behavior, 112–113
 - service specification contents, 125
 - service specification documentation, 420–421
 - service specification example, 129–130
 - <<use>> relationship, dependencies, 112
 - Utilization scenarios. *See* Usage scenarios

V

- ValidateProductID() operation, case study, 99–100
- Variant business responses, services, 252–257
- Version number scheme
 - OSGi, 153–154
 - WSDLs and XML schemas, 159–160, 164
- Versioning
 - architecture patterns for, 165–167
 - dependencies and compatibility, 152
 - determining number of versions, 169–170
 - incompatible changes, 163–164
 - objectives, 151–152
 - OSGi, 153–156
 - packages and, 152–153
 - SOAP interface addresses and, 168–169
 - summary review, 171–172
 - using major version numbers in names, 191
- Versioning, WSDL and XML schemas
 - backwards-compatible, 160–163
 - overview of, 156–157
 - version number placement for, 159–160
 - WSDL scope, 157–158
 - XML schema scope, 158
- Views
 - essential architecture, 4–7
 - TIBCO Collaborative Information Manager, 65
- Virtual machines, Intra-Site Cluster Failover Pattern using, 376
- Virtualization bindings, Service Bus load distribution, 272
- Virtualization thread pool, ActiveMatrix nodes, 343
- Virtualize policy set, thread usage with, 351–353

W

- Windows, TIBCO Adapter for IBM i for, 61
- Wiring, composite, 294
- Worker thread assignments
 - ActiveMatrix nodes, 345
 - ActiveMatrix BusinessWorks Service Engine thread usage, 361–364
 - default JMS thread usage, 347–349
 - default node-to-node communications thread usage, 349–351
 - default SOAP/HTTP thread usage, 345–347
 - thread usage with threading policy set, 354–356
 - thread usage with virtualize policy set, 351–353
- Write performance, disk limits, 331–334
- WSDL (Web Services Description Language)
 - backwards-compatible, 161–163
 - incompatible changes in, 163–164

- incorporating major version numbers in names, 191
- interface definitions, 176–178
- limiting scope, 157–158
- naming principles. *See* Naming principles
- organizing schema and interfaces for data model, 233–234
- rules for versioning, 164
- things requiring names, 182
- using distinct namespace URI for each interface, 190
- version number placement for, 159–160
- versioning, 156–157
- WSDL (Web Services Description Language) and Schemas
 - address schema, 444–445
 - carrier schema, 445–446
 - customer schema, 446–447
 - manufacturer schema, 447–448
 - other fulfillment schema, 449–450
 - phone schema, 448
 - product schema, 448–449
 - Sales Order Interface schema, 450–451
 - Sales Order Interface WSDL, 443–444
 - Sales Order schema, 451–452

X

- XML schemas
 - backwards-compatible, 160–163
 - incompatible changes in, 163
 - rules for versioning, 164
 - rules for WSDLs and schemas, 164
 - scope for imported, 158
 - version number placement for, 159–160
 - versioning, 156–157
- XML schemas, designing
 - creating information model, 229–232
 - creating schema, 232
 - identifying required entities and associations, 228–229
 - overview of, 227–228
- XSD (XML Schema Definition)
 - creating XML schema, 232
 - incorporating major version numbers in names, 191
 - naming principles for namespace URIs, 194–195
 - schema and interfaces for data model, 233–234
 - things requiring names, 183
 - WSDL interface definitions, 178

Z

- Z/OS platforms, 60–63