

SOA Design Patterns

*"This obligatory almanac of SOA design patterns
will become the foundation upon which many
organizations will build successful SOA solutions."*

*—Stephen Bennett, Director,
Technology Business Unit, Oracle*

 PRENTICE
HALL

With contributions by David Chappell, Jason Hogg, Anish Karmarkar, Mark Little, David Orchard,
Thomas Rischbeck, Satadru Roy, Arnaud Simon, Clemens Utschig, Dennis Wisnosky, and others.

Thomas Erl

Foreword by Grady Booch

FREE SAMPLE CHAPTER

SHARE WITH OTHERS







Agnostic Capability (324) How can multi-purpose service logic be made effectively consumable and composable?



Agnostic Context (312) How can multi-purpose service logic be positioned as an effective enterprise resource?



Agnostic Sub-Controller (607) How can agnostic, cross-entity composition logic be separated, reused, and governed independently?



Asynchronous Queuing (582) How can a service and its consumers accommodate isolated failures and avoid unnecessarily locking resources?



Atomic Service Transaction (623) How can a transaction with rollback capability be propagated across messaging-based services?



Brokered Authentication (661) How can a service efficiently verify consumer credentials if the consumer and service do not trust each other or if the consumer requires access to multiple services?



Canonical Expression (275) How can service contracts be consistently understood and interpreted?



Canonical Protocol (150) How can services be designed to avoid protocol bridging?



Canonical Resources (237) How can unnecessary infrastructure resource disparity be avoided?



Canonical Schema (158) How can services be designed to avoid data model transformation?

Canonical Schema Bus (709)



Canonical Versioning (286) How can service contracts within the same service inventory be versioned with minimal impact?



Capability Composition (521) How can a service capability solve a problem that requires logic outside of the service boundary?



Capability Recomposition (526) How can the same capability be used to help solve multiple problems?



Compatible Change (465) How can a service contract be modified without impacting consumers?



Compensating Service Transaction (631) How can composition runtime exceptions be consistently accommodated without requiring services to lock resources?



Composition Autonomy (616) How can compositions be implemented to minimize loss of autonomy?



Concurrent Contracts (421) How can a service facilitate multi-consumer coupling requirements and abstraction concerns at the same time?



Contract Centralization (409) How can direct consumer-to-implementation coupling be avoided?



Contract Denormalization (414) How can a service contract facilitate consumer programs with differing data exchange requirements?



Cross-Domain Utility Layer (267) How can redundant utility logic be avoided across domain service inventories?



Data Confidentiality (641) How can data within a message be protected so that it is not disclosed to unintended recipients while in transit?



Data Format Transformation (681) How can services interact with programs that communicate with different data formats?



Data Model Transformation (671) How can services interoperate when using different data models for the same type of data?



Data Origin Authentication (649) How can a service verify that a message originates from a known sender and that the message has not been tampered with in transit?



Decomposed Capability (504) How can a service be designed to minimize the chances of capability logic deconstruction?



Decoupled Contract (401) How can a service express its capabilities independently of its implementation?

Praise for this Book

“With the continued explosion of services and the increased rate of adoption of SOA through the market, there is a critical need for comprehensive, actionable guidance that provides the fastest possible time to results. Microsoft is honored to contribute to the *SOA Design Patterns* book, and to continue working with the community to realize the value of Real World SOA.”

- *Steven Martin, Senior Director, Developer Platform Product Management, Microsoft*

“*SOA Design Patterns* provides the proper guidance with the right level of abstraction to be adapted to each organization’s needs, and Oracle is pleased to have contributed to the patterns contained in this book.”

- *Dr. Mohamad Afshar, Director of Product Management, Oracle Fusion Middleware, Oracle*

“Red Hat is pleased to be involved in the *SOA Design Patterns* book and contribute important SOA design patterns to the community that we and our customers have used within our own SOA platforms. I am sure this will be a great resource for future SOA practitioners.”

- *Pierre Fricke Director, Product Line Management, JBoss SOA Platform, Red Hat*

“A wealth of proven, reusable SOA design patterns, clearly explained and illustrated with examples. An invaluable resource for all those involved in the design of service-oriented solutions.”

- *Phil Thomas, Consulting IT Specialist, IBM Software Group*

“This obligatory almanac of SOA design patterns will become the foundation on which many organizations will build their successful SOA solutions. It will allow organizations to build their own focused SOA design patterns catalog in an expedited fashion knowing that it contains the wealth and expertise of proven SOA best practices.”

- *Stephen Bennett, Director, Technology Business Unit, Oracle Corporation*

“The technical differences between service orientation and object orientation are subtle enough to confuse even the most advanced developers. Thomas Erl’s book provides a great service by clearly articulating SOA design patterns and differentiating them from similar OO design patterns.”

- *Anne Thomas Manes, VP & Research Director, Burton Group*

“*SOA Design Patterns* does an excellent job of laying out and discussing the areas of SOA design that a competent SOA practitioner should understand and employ.”

- *Robert Laird, SOA Architect, IBM*

“As always, Thomas delivers again. In a well-structured and easy-to-understand way, this book provides a wonderful collection of patterns each addressing a typical set of SOA design problems with well articulated solutions. The plain language and hundreds of diagrams included in the book help make the complicated subjects of SOA design comprehensible even to those who are new to the SOA design world. It’s a must-have reference book for all SOA practitioners, especially for enterprise architects, solution architects, developers, managers, and business process experts.”

- *Canyang Kevin Liu, Solution Architecture Manager, SAP*

“The concept of service oriented architecture has long promised visions of agile organizations being able to swap out interfaces and applications as business needs change. SOA also promises incredible developer and IT productivity, with the idea that key services would be candidates for cross-enterprise sharing or reuse. But many organizations’ efforts to move to SOA have been mired—by organizational issues, by conflicting vendor messages, and by architectures that may amount to little more than Just a Bunch of Web Services. There’s been a lot of confusion in the SOA marketplace about exactly what SOA is, what it’s supposed to accomplish, and how an enterprise goes about in making it work.

SOA Design Patterns is a definitive work that offers clarity on the purpose and functioning of service oriented architecture. *SOA Design Patterns* not only helps the IT practitioner lay the groundwork for a well-functioning SOA effort across the enterprise, but also connects the dots between SOA and the business requirements in a very concrete way. Plus, this book is completely technology agnostic—*SOA Design Patterns* rightly focuses on infrastructure and architecture, and it doesn’t matter whether you’re using components of one kind or another, or Java, or .NET, or Web services, or REST-style interfaces.

While no two SOA implementations are alike, Thomas Erl and his team of contributors have effectively identified the similarities in composition services need to have at a sub-atomic level in order to interact with each other as we hope they will. The book identifies 85 SOA design patterns which have been developed and thoroughly vetted to ensure that a service-oriented architecture does achieve the flexibility and loose coupling promised. The book is also compelling in that it is a living document, if you will, inviting participation in an open process to identify and formulate new patterns to this growing body of knowledge.”

- *Joe McKendrick, Independent Analyst, Author of ZDNet’s SOA Blog*

“If you want to truly educate yourself on SOA, read this book.”

- *Sona Srinivasan, Global Client Services & Operations, CISCO*

“An impressive decomposition of the process and architectural elements that support service-oriented analysis, design, and delivery. Right-sized and terminologically consistent.

Overall, the book represents a patient separation of concerns in respect of the process and architectural parts that underpin any serious SOA undertaking. Two things stand out. First, the pattern relationship diagrams provide rich views into the systemic relationships that structure a service-oriented architecture: these patterns are not discrete, isolated templates to be applied

mechanically to the problem space; rather, they form a network of forces and constraints that guide the practitioner to consider the task at hand in the context of its inter-dependencies. Second, the pattern sequence diagrams and accompanying notes provide a useful framework for planning and executing the many activities that comprise an SOA engagement.”

- Ian Robinson, *Principal Technology Consultant, ThoughtWorks*

“Successful implementation of SOA principles requires a shift in focus from software system means, or the way capabilities are developed, to the desired end results, or real-world effects required to satisfy organizational business processes. In *SOA Design Patterns*, Thomas Erl provides service architects with a broad palette of reusable service patterns that describe service capabilities that can cut across many SOA applications. Service architects taking advantage of these patterns will save a great deal of time describing and assembling services to deliver the real world effects they need to meet their organization’s specific business objectives.”

- Chuck Georgo, *Public Safety and National Security Architect*

“In IT, we have increasingly come to see the value of having catalogs of good solution patterns in programming and systems design. With this book, Thomas Erl brings a comprehensive set of patterns to bear on the world of SOA. These patterns enable easily communicated, reusable, and effective solutions, allowing us to more rapidly design and build out the large, complicated and interoperable enterprise SOAs into which our IT environments are evolving.”

- Al Gough, *Business Systems Solutions CTO, CACI International Inc.*

“This book provides a comprehensive and pragmatic review of design issues in service-centric design, development, and evolution. The Web site related to this book [SOAPatterns.org] is a wonderful platform and gives the opportunity for the software community to maintain this catalogue....”

- Veronica Gacitua Decar, *Dublin City University*

“Erl’s *SOA Design Patterns* is for the IT decision maker determined to make smart architecture design choices, smart investments, and long term enterprise impact. For those IT professionals committed to service-orientation as a value-added design and implementation option, *Patterns* offers a credible, repeatable approach to engineering an adaptable business enterprise. This is a must read for all IT architect professionals.”

- Larry Gloss, *VP and General Manager, Information Manufacturing, LLC*

“These SOA patterns define, encompass, and comprise a complete repertoire of best practices for developing a world-class IT SOA portfolio for the enterprise and its organizational units through to service and schema analysis and design. After many years as an architect on many SOA projects, I strongly recommend this book be on the shelf of every analyst and technical member of any SOA effort, right next to the SOA standards and guidelines it outlines and elucidates the need for. Our SOA governance standards draw heavily from this work and others from this series.”

- Robert John Hathaway III, *Enterprise Software Architect, SOA Object Systems*

“A wise man once told me that wisdom isn’t all about knowledge and intelligence, it is just as much about asking questions. Asking questions is the true mark of wisdom and during the writing of the *SOA Design Patterns* book Thomas Erl has shown his real qualities. The community effort behind this book is huge meaning that Thomas has had access to the knowledge and experience of a large group of accomplished practitioners. The result speaks for itself. This book is packed with proven solutions to recurring problems, and the documented pros and cons of each solution have been verified by persons with true experience. This book could give SOA initiatives of any scale a real boost.”

- *Herbjörn Wilhelmsen, Architect and Senior Consultant, Objectware*

“This book is an absolute milestone in SOA literature. For the first time we are provided with a practical guide on how the principle centric description of service orientation from a vendor-agnostic viewpoint is actually made to work in a language based on patterns. This book makes you talk SOA! There are very few who understand SOA like Thomas Erl does, he actually put’s it all together!”

- *Brian Lokhorst, Solution Architect, Dutch Tax Office*

“Service oriented architecture is all about best practices we have learned since IT’s existence. This book takes all those best practices and bundles them into a nice pattern catalogue. [It provides] a really excellent approach as patterns are not just documented but are provided with application scenarios through case studies [which] fills the gap between theory and practice.”

- *Shakti Sharma, Senior Enterprise Architect, Sysco Corp*

“An excellent and important book on solving problems in SOA [with a] solid structure. Has the potential of being among the major influential books.”

- *Peter Chang, Lawrence Technical University*

“*SOA Design Patterns* presents a vast amount of knowledge about how to successfully implement SOA within an organization. The information is clear, concise, and most importantly, legitimate.”

- *Peter B. Woodhull, President and Principal Architect, Modus21*

“*SOA Design Patterns* offers real insights into everyday problems that one will encounter when investing in services oriented architecture. [It] provides a number of problem descriptions and offers strategies for dealing with these problems. SOA design patterns highlights more than just the technical problems and solutions. Common organizational issues that can hinder progress towards achieving SOA migration are explained along with potential approaches for dealing with these real world challenges. Once again Thomas Erl provides in-depth coverage of SOA terminology and helps the reader better understand and appreciate the complexities of migrating to an SOA environment.”

- *David Michalowicz, Air and Space Operations Center Modernization Team Lead, MITRE Corporation*

“This is a long overdue, serious, comprehensive, and well-presented catalog of SOA design patterns. This will be required reading and reference for all our SOA engineers and architects. The best of the series so far!

[The book] works in two ways: as a primer in SOA design and architecture it can easily be read front-to-back to get an overview of most of the key design issues you will encounter, and as a reference catalog of design techniques that can be referred to again and again ...”

- *Wendell Ocasio, Architecture Consultant, DoD Military Health Systems, Agilex Technologies*

“Thomas has once again provided the SOA practitioner with a phenomenal collection of knowledge. This is a reference that I will come back to time and time again as I move forward in SOA design efforts.

What I liked most about this book is its vendor agnostic approach to SOA design patterns. This approach really presents the reader with an understanding of why or why not to implement a pattern, group patterns, or use compound patterns rather than giving them a marketing spiel on why one implementation of a pattern is better than another (for example, why one ESB is better than another). I think as SOA adoption continues to advance, the ability for architects to understand when and why to apply specific patterns will be a driving factor in the overall success and evolution of SOA. Additionally, I believe that this book provides the consumer with the understanding required to chose which vendor’s SOA products are right for their specific needs.”

- *Bryan Brew, SOA Consultant, Booz Allen Hamilton*

“A must have for every SOA practitioner.”

- *Richard Van Schelven, Principal Engineer, Ericsson*

“This book is a long-expected successor to the books on object-oriented design patterns and integration patterns. It is a great reference book that clearly and thoroughly describes design patterns for SOA. A great read for architects who are facing the challenge of transforming their enterprise into a service-oriented enterprise.”

- *Linda Terlouw, Solution Architect, Ordina*

“The maturation of Service-Oriented Architecture has given the industry time to absorb the best practices of service development. Thomas Erl has amassed this collective wisdom in *SOA Design Patterns*, an absolutely indispensable addition to any Service Oriented bookshelf.”

- *Kevin P. Davis, Ph.D*

“The problem with most texts on SOA is one of specificity. Architects responsible for SOA implementation in most organizations have little time for abstract theories on the subject, but are hungry for concrete details that they can relate to the real problems they face in their environment. *SOA Design Patterns* is critical reading for anyone with service design responsibilities. Not only does the text provide the normal pattern templates, but each pattern is applied in

detail against a background case study to provide exceptionally meaningful context to the information. The graphic visualizations of the problems and pattern solutions are excellent supplementary companions to the explanatory text. This book will greatly stretch the knowledge of the reader as much for raising and addressing issues that may have never occurred to the reader as it does in treating those problems that are in more common occurrence. The real beauty of this book is in its plain English prose. Unlike so many technical reference books, one does not find themselves re-reading sections multiple times trying to discern the intent of the author. This is also not a reference that will sit gathering dust on a shelf after one or two perusings. Practitioners will find themselves returning over and over to utilize the knowledge in their projects. This is as close as you'll come to having a service design expert sitting over your shoulder."

- James Kinneavy, *Principal Software Architect, University of California*

"As the industry converges on SOA patterns, Erl provides an outstanding reference guide to composition and integration—and yet another distinctive contribution to the SOA practice."

- Steve Birkel, *Chief IT Technical Architect, Intel Corp.*

"With *SOA Design Patterns*, Thomas Erl adds an indispensable SOA reference volume to the technologist's library. Replete with to-the-point examples, it will be a helpful aid to any IT organization."

- Ed Dodds, *Strategist, Systems Architect, Conmergence*

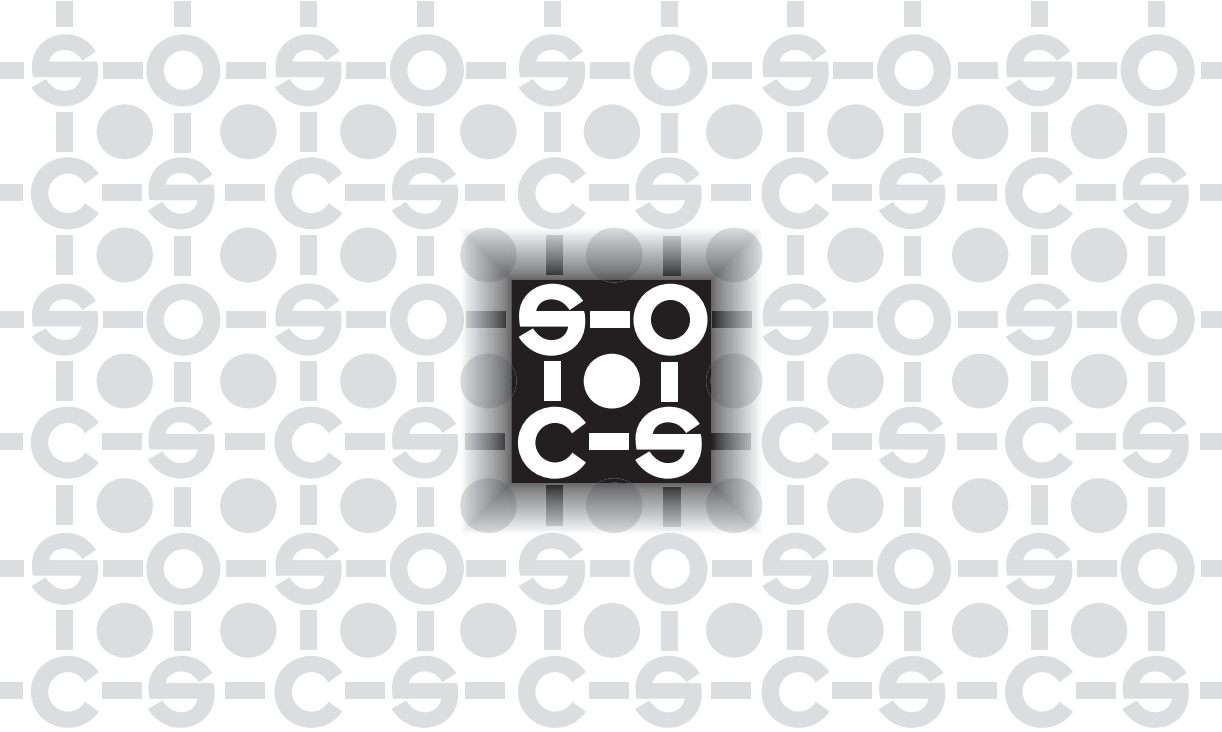
"Again, Thomas Erl has written an indispensable guide to SOA. Building on his prior successes, his patterns go into even more detail. Therefore, this book is not only helpful to the SOA beginner, but also provides new insight and ideas to professionals."

- Philipp Offermann, *Research Scientist, Technische Universität Berlin, Germany*

"*SOA Design Patterns* is an extraordinary contribution to SOA best practices! Once again, Thomas has created an indispensable resource for any person or organization interested in or actively engaged in the practice of Service Oriented Architecture. Using case studies based on three very different business models, Thomas guides the reader through the process of selecting appropriate implementation patterns to ensure a flexible, well-performing, and secure SOA ecosystem."

- Victor Brown, *Managing Partner and Principal Consultant,
Cypress Management Group Corporation*

SOA Design Patterns



The Prentice Hall Service-Oriented Computing Series from Thomas Erl aims to provide the IT industry with a consistent level of unbiased, practical, and comprehensive guidance and instruction in the areas of service-oriented architecture, service-orientation, and the expanding landscape that is shaping the real-world service-oriented computing platform.

For more information, visit www.soabooks.com.

SOA Design Patterns

Thomas Erl

(with additional contributors)



PRENTICE HALL

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.

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

Library of Congress Cataloging-in-Publication Data:

Erl, Thomas.

SOA design patterns / Thomas Erl. — 1st ed.

p. cm.

ISBN 0-13-613516-1 (hardback : alk. paper) 1. Web services.

2. Computer architecture. 3. Software patterns. 4. System design. I. Title.

TK5105.88813.E735 2008

006.7—dc22

2008040488

Copyright © 2009 SOA Systems 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. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN-13: 978-0-13-613516-6

ISBN-10: 0-13-613516-1

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.

Fifth Printing: April 2013

The following patterns: Exception Shielding, Threat Screening, Trusted Subsystem, Service Perimeter Guard, Data Confidentiality, Data Origin Authentication, Direct Authentication, Brokered Authentication are courtesy of the Microsoft Patterns & Practices team. For more information please visit <http://msdn.microsoft.com/practices>.

These patterns were originally developed by Jason Hogg, Frederick Chong, Dwayne Taylor, Lonnie Wall, Paul Slater, Tom Hollander, Wojtek Kozaczynski, Don Smith, Larry Brader, Sajjad Nasir Imran, Pablo Cibraro, Nelly Delgado and Ward Cunningham

Editor-in-Chief

Mark L. Taub

Managing Editor

Kristy Hart

Copy Editor

Language Logistics

Indexer

Cheryl Lenser

Proofreader

Williams Woods
Publishing

Composition

Jake McFarland
Bumpy Design

Graphics

Zuzana Cappova
Tami Young
Spencer Fruhling

Photos

Thomas Erl

Cover Design

Thomas Erl

To the SOA pioneers that blazed the trail we now so freely base our roadmaps on, and to the SOA community that helped me refine the wisdom of the pioneers into this catalog of patterns.

- Thomas Erl

This page intentionally left blank

Contents

Foreword	xxxvii
-----------------------	---------------

CHAPTER 1: Introduction	1
--------------------------------------	----------

1.1 Objectives of this Book	4
1.2 Who this Book is For	4
1.3 What this Book Does Not Cover	4
Topics Covered by Other Books	4
Web Service and REST Service Design Patterns	5
SOA Standardization Efforts	5
1.4 Recommended Reading	6
1.5 How this Book is Organized	7
Part I: Fundamentals	8
Part II: Service Inventory Design Patterns	8
Part III: Service Design Patterns	8
Part IV: Service Composition Design Patterns	9
Part V: Supplemental	10
Part VI: Appendices	10
1.6 Symbols, Figures, Style Conventions	11
Symbol Legend	11
How Color is Used	11
Data Flow and Directionality Conventions	11
Pattern Documentation Conventions	11
1.7 Additional Information	11
Updates, Errata, and Resources (www.soabooks.com)	11
Visio Stencil (www.soabooks.com)	12

- Community Patterns Site (www.soapatterns.org) 12
- Master Glossary (www.soaglossary.com) 12
- Supplementary Posters (www.soaposters.com) 12
- The SOA Magazine (www.soamag.com) 12
- Referenced Specifications (www.soaspecs.com). 12
- Notification Service 13
- Contact the Author 13

CHAPTER 2: Case Study Background 15

- 2.1 Case #1 Background: Cutit Saws Ltd. 17
 - History 18
 - Technical Infrastructure and Automation Environment 18
 - Business Goals and Obstacles 18
- 2.2 Case #2 Background: Alleywood Lumber Company 19
 - History 19
 - Technical Infrastructure and Automation Environment 20
 - Business Goals and Obstacles 20
- 2.3 Case #3 Background: Forestry Regulatory Commission (FRC) 21
 - History 21
 - Technical Infrastructure and Automation Environment 21
 - Business Goals and Obstacles 22

PART I: FUNDAMENTALS

CHAPTER 3: Basic Terms and Concepts 25

- Purpose of this Introductory Chapter 26
- 3.1 Architecture Fundamentals 26
 - A Classic Analogy for Architecture and Infrastructure 27
 - Technology Architecture 27
 - Technology Infrastructure 30
 - Software Program 32
 - Relationship to Design Framework 33

- 3.2 Service-Oriented Computing Fundamentals. 35
 - Service-Oriented Computing 35
 - Service-Orientation 36
 - Service-Oriented Architecture (SOA) 37
 - Service 37
 - Service Capability 38
 - Service Consumer 38
 - Service Composition 40
 - Service Inventory. 42
 - Service-Oriented Analysis. 43
 - Service Candidate. 44
- 3.3 Service Implementation Mediums. 44
 - Services as Components 45
 - Services as Web Services 45
 - REST Services. 46

CHAPTER 4: The Architecture of Service-Orientation . . 47

- Purpose of this Introductory Chapter 48
- 4.1 The Method of Service-Orientation 48
 - Principles of Service-Orientation. 48
 - Strategic Goals of Service-Oriented Computing. 51
- 4.2 The Four Characteristics of SOA. 52
 - Business-Driven 53
 - Vendor-Neutral 54
 - Enterprise-Centric 58
 - Composition-Centric 59
- 4.3 The Four Common Types of SOA 61
 - Service Architecture 62
 - Information Hiding* 64
 - Design Standards* 64
 - Service Contracts* 65
 - Service Agents* 67
 - Service Capabilities* 68
 - Service Composition Architecture 68
 - Nested Compositions* 72
 - Task Services and Alternative Compositions* 73
 - Compositions and Infrastructure.* 74

- Service Inventory Architecture 74
- Service-Oriented Enterprise Architecture 76
- Architecture Types and Scope 77
- Architecture Types and Inheritance 77
- Other Forms of Service-Oriented Architecture 78
 - Inter-Business Service Architecture* 78
 - Service-Oriented Community Architecture* 78
- 4.4 The End Result of Service-Orientation 79

CHAPTER 5: Understanding SOA Design Patterns 85

- Purpose of this Introductory Chapter 86
- 5.1 Fundamental Terminology 86
 - What's a Design Pattern? 86
 - What's a Compound Pattern? 88
 - What's a Design Pattern Language? 88
 - What's a Design Pattern Catalog? 89
- 5.2 Historical Influences 89
 - Alexander's Pattern Language 90
 - Object-Oriented Patterns 91
 - Software Architecture Patterns 92
 - Enterprise Application Architecture Patterns 93
 - EAI Patterns 93
 - SOA Patterns 94
- 5.3 Pattern Notation 95
 - Pattern Symbols 95
 - Pattern Figures 96
 - Pattern Application Sequence Figures* 96
 - Pattern Relationship Figures* 96
 - Compound Pattern Hierarchy Figures* 99
 - Capitalization 100
 - Page Number References 100
- 5.4 Pattern Profiles 100
 - Requirement 101
 - Icon 101
 - Summary 102
 - Problem 102
 - Solution 102

- Application 103
- Impacts 103
- Relationships 103
- Case Study Example 103
- 5.5 Patterns with Common Characteristics 104
 - Canonical Patterns 104
 - Centralization Patterns 105
- 5.6 Key Design Considerations. 106
 - “Enterprise” vs. “Enterprise-wide” 106
 - Design Patterns and Design Principles 106
 - Design Patterns and Design Granularity. 107
 - Measures of Design Pattern Application. 108

PART II: SERVICE INVENTORY DESIGN PATTERNS

CHAPTER 6: Foundational Inventory Patterns 111

- How Inventory Design Patterns Relate to SOA Design
 - Characteristics 113
 - How Foundational Inventory and Service Patterns Relate 114
 - How Case Studies are Used in this Chapter. 114
- 6.1 Inventory Boundary Patterns. 114
- Enterprise Inventory 116**
 - Problem 116
 - Solution 117
 - Application 118
 - Impacts 120
 - Relationships 121
 - Case Study Example 122
- Domain Inventory 123**
 - Problem 123
 - Solution 124
 - Application 125
 - Impacts 126
 - Relationships 127
 - Case Study Example 128

- 6.2 Inventory Structure Patterns 130
- Service Normalization 131**
 - Problem 131
 - Solution 132
 - Application 132
 - Impacts 133
 - Relationships 133
 - Case Study Example 135
- Logic Centralization 136**
 - Problem 136
 - Solution 137
 - Application 137
 - Impacts 139
 - Relationships 140
 - Case Study Example 142
- Service Layers 143**
 - Problem 143
 - Solution 144
 - Application 145
 - Impacts 147
 - Relationships 147
 - Case Study Example 148
- 6.3 Inventory Standardization Patterns 149
- Canonical Protocol 150**
 - Problem 151
 - Solution 152
 - Application 153
 - Impacts 155
 - Relationships 155
 - Case Study Example 157
- Canonical Schema 158**
 - Problem 158
 - Solution 159
 - Application 159
 - Impacts 159
 - Relationships 160
 - Case Study Example 161

CHAPTER 7: Logical Inventory Layer Patterns 163

Combining Layers 164
Business Logic and Utility Logic 166
Agnostic Logic and Non-Agnostic Logic 166
Service Layers and Logic Types 167

Utility Abstraction 168

Problem 168
Solution 169
Application 170
Impacts 171
Relationships 171
Case Study Example 173

Entity Abstraction 175

Problem 175
Solution 176
Application 176
Impacts 178
Relationships 178
Case Study Example 180

Process Abstraction 182

Problem 182
Solution 183
Application 184
Impacts 185
Relationships 185
Case Study Example 187

CHAPTER 8: Inventory Centralization Patterns 191

Process Centralization 193

Problem 193
Solution 194
Application 195
Impacts 196
Relationships 197
Case Study Example 198

- Schema Centralization 200**
 - Problem 200
 - Solution 201
 - Application 202
 - Impacts 202
 - Relationships 203
 - Case Study Example 203

- Policy Centralization 207**
 - Problems 207
 - Solution 208
 - Application 209
 - Impacts 210
 - Relationships 211
 - Case Study Example 213

- Rules Centralization 216**
 - Problem 216
 - Solution 217
 - Application 217
 - Impacts 218
 - Relationships 219
 - Case Study Example 222

CHAPTER 9: Inventory Implementation Patterns 225

- Dual Protocols 227**
 - Problem 228
 - Solution 228
 - Application 228
 - Impacts 233
 - Relationships 234
 - Case Study Example 235

- Canonical Resources 237**
 - Problem 238
 - Solution 238
 - Application 239
 - Impacts 239

- Relationships 239
- Case Study Example 241
- State Repository 242**
 - Problem 242
 - Solution 243
 - Application 244
 - Impacts 244
 - Relationships 244
 - Case Study Example 246
- Stateful Services 248**
 - Problem 248
 - Solution 248
 - Application 250
 - Impacts 250
 - Relationships 250
 - Case Study Example 251
- Service Grid 254**
 - Problem 254
 - Solution 255
 - Application 256
 - Impacts 257
 - Relationships 258
 - Case Study Example 259
- Inventory Endpoint 260**
 - Problem 260
 - Solution 261
 - Application 262
 - Impacts 263
 - Relationships 263
 - Case Study Example 265
- Cross-Domain Utility Layer 267**
 - Problem 267
 - Solution 268
 - Application 269
 - Impacts 269
 - Relationships 270
 - Case Study Example 270

CHAPTER 10: Inventory Governance Patterns 273

- Canonical Expression 275**
 - Problem 275
 - Solution 275
 - Application 276
 - Impacts 277
 - Relationships 278
 - Case Study Example 279
- Metadata Centralization 280**
 - Problem 280
 - Solution 281
 - Application 282
 - Impacts 283
 - Relationships 283
 - Case Study Example 284
- Canonical Versioning 286**
 - Problem 286
 - Solution 287
 - Application 287
 - Impacts 288
 - Relationships 288
 - Case Study Example 290

PART III: SERVICE DESIGN PATTERNS

CHAPTER 11: Foundational Service Patterns 295

- Case Study Background 297
- 11.1 Service Identification Patterns 299
- Functional Decomposition 300**
 - Problem 300
 - Solution 301
 - Application 302
 - Impacts 302
 - Relationships 303
 - Case Study Example 303

Service Encapsulation 305

- Problem 305
- Solution 306
- Application 307
- Impacts 309
- Relationships 309
- Case Study Example 310

11.2 Service Definition Patterns 311

Agnostic Context. 312

- Problem 313
- Solution 314
- Application 315
- Impacts 315
- Relationships 316
- Case Study Example 317

Non-Agnostic Context 319

- Problem 319
- Solution 320
- Application 321
- Impacts 322
- Relationships 322
- Case Study Example 323

Agnostic Capability. 324

- Problem 324
- Solution 325
- Application 326
- Impacts 327
- Relationships 327
- Case Study Example 328

CHAPTER 12: Service Implementation Patterns. 331

Service Façade 333

- Problem 333
- Solution 334
- Application 335

- Impacts 341
- Relationships 342
- Case Study Example 343
- Redundant Implementation 345**
 - Problem 345
 - Solution 346
 - Application 346
 - Impacts 347
 - Relationships 348
 - Case Study Example 349
- Service Data Replication 350**
 - Problem 350
 - Solution 352
 - Application 353
 - Impacts 353
 - Relationships 353
 - Case Study Example 354
- Partial State Deferral 356**
 - Problem 356
 - Solution 357
 - Application 358
 - Impacts 359
 - Relationships 359
 - Case Study Example 360
- Partial Validation 362**
 - Problem 362
 - Solution 363
 - Application 364
 - Impacts 364
 - Relationships 364
 - Case Study Example 365
- UI Mediator 366**
 - Problem 366
 - Solution 367
 - Application 368

Impacts 369
Relationships 370
Case Study Example 370

CHAPTER 13: Service Security Patterns 373

Case Study background 374

Exception Shielding 376

 Problem 376
 Solution 377
 Application 378
 Impacts 379
 Relationships 379
 Case Study Example 380

Message Screening 381

 Problem 381
 Solution 382
 Application 382
 Impacts 384
 Relationships 385
 Case Study Example 385

Trusted Subsystem 387

 Problem 387
 Solution 388
 Application 388
 Impacts 391
 Relationships 391
 Case Study Example 392

Service Perimeter Guard 394

 Problem 394
 Solution 395
 Application 395
 Impacts 396
 Relationships 396
 Case Study Example 397

CHAPTER 14: Service Contract Design Patterns 399

- Decoupled Contract 401**
 - Problem 401
 - Solution 402
 - Application 403
 - Impacts 405
 - Relationships 405
 - Case Study Example 407
- Contract Centralization 409**
 - Problem 409
 - Solution 410
 - Application 410
 - Impacts 411
 - Relationships 411
 - Case Study Example 413
- Contract Denormalization 414**
 - Problem 414
 - Solution 415
 - Application 416
 - Impacts 417
 - Relationships 417
 - Case Study Example 418
- Concurrent Contracts 421**
 - Problem 421
 - Solution 422
 - Application 423
 - Impacts 425
 - Relationships 425
 - Case Study Example 426
- Validation Abstraction 429**
 - Problem 429
 - Solution 430
 - Application 431
 - Impacts 432
 - Relationships 432
 - Case Study Example 433

Chapter 15: Legacy Encapsulation Patterns 439

- Legacy Wrapper. 441**
 - Problem 441
 - Solution 442
 - Application 443
 - Impacts 444
 - Relationships 444
 - Case Study Example 446
- Multi-Channel Endpoint 451**
 - Problem 451
 - Solution 452
 - Application 453
 - Impacts 454
 - Relationships 454
 - Case Study Example 456
- File Gateway 457**
 - Problem 457
 - Solution 458
 - Application 458
 - Impacts 459
 - Relationships 460
 - Case Study Example 461

CHAPTER 16: Service Governance Patterns. 463

- Compatible Change. 465**
 - Problem 465
 - Solution 466
 - Application 466
 - Impacts 469
 - Relationships 469
 - Case Study Example 470
- Version Identification 472**
 - Problem 472
 - Solution 473
 - Application 473

- Impacts 474
- Relationships 474
- Case Study Example 475
- Termination Notification 478**
- Problem 478
- Solution 479
- Application 480
- Impacts 480
- Relationships 481
- Case Study Example 481
- Service Refactoring. 484**
- Problem 484
- Solution 485
- Application 485
- Impacts 486
- Relationships 486
- Case Study Example 488
- Service Decomposition. 489**
- Problem 489
- Solution 491
- Application 492
- Impacts 492
- Relationships 494
- Case Study Example 495
- Proxy Capability 497**
- Problem 497
- Solution 498
- Application 498
- Impacts 500
- Relationships 500
- Case Study Example 501
- Decomposed Capability 504**
- Problem 504
- Solution 506
- Application 507
- Impacts 507

- Relationships 508
- Case Study Example 508
- Distributed Capability 510**
- Problem 510
- Solution 511
- Application 512
- Impacts 513
- Relationships 513
- Case Study Example 514

PART IV: SERVICE COMPOSITION DESIGN PATTERNS

CHAPTER 17: Capability Composition Patterns 519

- Capability Composition 521**
- Problem 521
- Solution 521
- Application 523
- Impacts 523
- Relationships 523
- Case Study Example 524
- Capability Recomposition 526**
- Problem 526
- Solution 527
- Application 527
- Impacts 527
- Relationships 529
- Case Study Example 530

CHAPTER 18: Service Messaging Patterns 531

- Service Messaging 533**
- Problem 533
- Solution 533
- Application 534
- Impacts 534

- Relationships 535
- Case Study Example 536
- Messaging Metadata 538**
 - Problem 538
 - Solution 538
 - Application 539
 - Impacts 540
 - Relationships 541
 - Case Study Example 542
- Service Agent 543**
 - Problem 543
 - Solution 544
 - Application 544
 - Impacts 546
 - Relationships 546
 - Case Study Example 548
- Intermediate Routing 549**
 - Problem 549
 - Solution 551
 - Application 552
 - Impacts 553
 - Relationships 553
 - Case Study Example 556
- State Messaging 557**
 - Problem 557
 - Solution 558
 - Application 560
 - Impacts 561
 - Relationships 561
 - Case Study Example 562
- Service Callback 566**
 - Problem 566
 - Solution 568
 - Application 568
 - Impacts 570

Relationships 570

Case Study Example 571

Service Instance Routing 574

 Problem 574

 Solution 576

 Application 576

 Impacts 578

 Relationships 578

 Case Study Example 579

Asynchronous Queuing 582

 Problem 582

 Solution 584

 Application 584

 Impacts 587

 Relationships 588

 Case Study Example 589

Reliable Messaging 592

 Problem 592

 Solution 593

 Application 593

 Impacts 594

 Relationships 595

 Case Study Example 596

Event-Driven Messaging 599

 Problem 599

 Solution 600

 Application 602

 Impacts 602

 Relationships 602

 Case Study Example 604

CHAPTER 19: Composition Implementation Patterns . . 605

Agnostic Sub-Controller 607

 Problem 607

 Solution 608

- Application 610
- Impacts 610
- Relationships 610
- Case Study Example 612
- Composition Autonomy 616**
 - Problem 616
 - Solution 618
 - Application 619
 - Impacts 619
 - Relationships 620
 - Case Study Example 620
- Atomic Service Transaction 623**
 - Problem 623
 - Solution 624
 - Application 626
 - Impacts 626
 - Relationships 628
 - Case Study Example 629
- Compensating Service Transaction 631**
 - Problem 631
 - Solution 633
 - Application 633
 - Impacts 635
 - Relationships 635
 - Case Study Example 636
- CHAPTER 20: Service Interaction Security Patterns . . 639**
 - Data Confidentiality 641**
 - Problem 641
 - Solution 643
 - Application 643
 - Impacts 644
 - Relationships 645
 - Case Study Example 646

Data Origin Authentication	649
Problem	649
Solution	650
Application	651
Impacts	652
Relationships	653
Case Study Example	653
Direct Authentication	656
Problem	656
Solution	657
Application	657
Impacts	658
Relationships	659
Case Study Example	660
Brokered Authentication	661
Problem	661
Solution	662
Application	663
Impacts	665
Relationships	665
Case Study Example	666
CHAPTER 21: Transformation Patterns	669
Data Model Transformation	671
Problem	671
Solution	672
Application	673
Impacts	674
Relationships	674
Case Study Example	677
Data Format Transformation	681
Problem	681
Solution	681
Application	683

- Impacts 683
- Relationships 683
- Case Study Example 685
- Protocol Bridging 687**
 - Problem 687
 - Solution 688
 - Application 688
 - Impacts 690
 - Relationships 690
 - Case Study Example 692

PART V: SUPPLEMENTAL

CHAPTER 22: Common Compound Design Patterns . . . 697

- “Compound” vs. “Composite” 698
- Compound Patterns and Pattern Relationships 698
- Joint Application vs. Coexistent Application 699
- Compound Patterns and Pattern Granularity 700
- Orchestration 701**
- Enterprise Service Bus 704**
- Service Broker 707**
- Canonical Schema Bus 709**
- Official Endpoint 711**
- Federated Endpoint Layer 713**
- Three-Layer Inventory 715**

CHAPTER 23: Strategic Architecture Considerations . . 717

- Increased Federation 718
- Increased Intrinsic Interoperability 721
- Increased Vendor Diversification Options 723

Increased Business and Technology Alignment 725

Increased ROI 727

Increased Organizational Agility 728

Reduced IT Burden 729

CHAPTER 24: Principles and Patterns at the U.S. Department of Defense 731

The Business Operating Environment (BOE) 733

Principles, Patterns, and the BOE 734

 Incorporation of Information Assurance (IA) 736

 Adherence to Standards 736

 Data Visibility, Accessibility, and Understandability to Support Decision Makers 736

 Loosely Coupled Services 736

 Authoritative Sources of Trusted Data 737

 Metadata-Driven Framework for Separation from Technical Details 737

 Support Use of Open Source Software 738

 Emphasize Use of Service-Enabled Commercial Off-the-Shelf (COTS) Software 738

 Participation in the DoD Enterprise 738

 Support Mobility — Users & Devices 738

The Future of SOA and the DoD 739

SOADoD.org 739

PART VI: APPENDICES

APPENDIX A: Case Study Conclusion 743

Cutit Saws Ltd. 744

Alleywood Lumber Company 744

Forestry Regulatory Commission (FRC) 745

APPENDIX B: Candidate Patterns 747

APPENDIX C: Principles of Service-Orientation 749

- Standardized Service Contract 751
- Service Loose Coupling 753
- Service Abstraction 755
- Service Reusability 756
- Service Autonomy 758
- Service Statelessness 760
- Service Discoverability 762
- Service Composability 764

**APPENDIX D: Patterns and Principles
Cross-Reference 767**

**APPENDIX E: Patterns and Architecture Types
Cross-Reference 775**

About the Author 783

About the Contributors 784

Index of Patterns 791

Index 795

Foreword

The entire history of software engineering can be characterized as one of rising levels of abstraction. We see this in our languages, our tools, our platforms, and our methods. Indeed, abstraction is the primary way that we as humans attend to complexity—and software-intensive systems are among the most complex artifacts ever created.

I would also observe that one of the most important advances in software engineering over the past two decades has been the practice of patterns. Patterns are yet another example of this rise in abstraction: A pattern specifies a common solution to a common problem in the form of a society of components that collaborate with one another. Influenced by the writings of Christopher Alexander, Kent Beck and Ward Cunningham began to codify various design patterns from their experience with Smalltalk. Growing slowly but steadily, these concepts began to gain traction among other developers. The publication of the seminal book *Design Patterns* by Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm marked the introduction of these ideas to the mainstream. The subsequent activities of the Hillside Group provided a forum for this growing community, yielding a very vibrant literature and practice. Now the practice of patterns is very much mainstream: Every well-structured software-intensive system tends to be full of patterns (whether their architects name them intentionally or not).

The emerging dominant architectural style for many enterprise systems is that of a service-oriented architecture, a style that at its core is essentially a message passing architecture. However, therein are many patterns that work (and anti-patterns that should be avoided).

Thomas' work is therefore the right book at the right time. He really groks the nature of SOA systems: There are many hard design decisions to be made, ranging from data-orientation to the problems of legacy integration and even security. Thomas offers wise counsel on each of these issues and many more, all in the language of design patterns. There are many things I like about this work. It's comprehensive. It's written in a very accessible

pattern language. It offers patterns that play well with one another. Finally, Thomas covers not just the technical details, but also sets these patterns in the context of economic and other considerations.

SOA Design Patterns is an important contribution to the literature and practice of building and delivering quality software-intensive systems.

—*Grady Booch, IBM Fellow*
September, 2008

Acknowledgments

This book was in development for over three years, a good portion of which was dedicated to external reviews. Patterns were subjected to three review cycles that spanned a period of over twelve months and involved over 200 IT professionals. Pre-release galleys of my first and second manuscript drafts were printed and shipped to SOA experts and patterns experts around the world. Additionally, I had the full manuscript published at SOAPatterns.org for an open industry review. Even though these review phases added much time and effort to the development of this book, they ultimately elevated the quality of this work by a significant margin.

Special thanks to Prentice Hall for their patience and support throughout the book development process. Specifically, I'd like to thank Kristy Hart and Jake McFarland for their tremendous production efforts and tireless commitment to achieving printed perfection, Mark Taub who stood by this book project through a whirlwind of changes, reviews, more changes, extensions, and delays, Stephane Nakib and Heather Fox for their on-going guidance, and Eric Miller for his assistance with publishing the online review version of the first manuscript draft. I am fortunate to be working with the best publishing team in the industry.

Special thanks also to Herbjörn Wilhelmsen, Martin Fowler, Ralph Johnson, Bobby Woolf, Grady Booch, Gregor Hohpe, Baptist Eggen, Dragos Manolescu, Frank Buschmann, Wendell Ocasio, and Kevin Davis for their guidance and uninhibited feedback throughout the review cycles.

My thanks and gratitude to the following reviewers that participated in one or more of the manuscript reviews (in alphabetical order by last name):

Mohamad Afshar, Oracle

Sanjay Agara, Wipro

Stephen Bennett, Oracle

Steve Birkel, Intel

Brandon Bohling, Intel

Grady Booch, IBM
Bryan Brew, Booz Allen Hamilton
Victor Brown, CMGC
Frank Buschmann, Siemens
Enrique G. Castro-Leon, Intel
Peter Chang, Lawrence Technical University
Jason “AJ” Comfort, Booz Allen Hamilton
John Crupi, JackBe
Veronica Gacitua Decar, Dublin City University
Ed Dodds, Conmergence
Kevin P. Davis, PhD
Dominic Duggan, Stevens Institute of Technology
Baptist Eggen, Dutch Department of Defense
Steve Elston, Microsoft
Dale Ferrario, Sun Microsystems
Martin Fowler, ThoughtWorks
Pierre Fricke, Red Hat
Chuck Georgo, Public Safety and National Security
Larry Gloss, Information Manufacturing
Al Gough, CACI International Inc.
Daniel Gross, University of Toronto
Robert John Hathaway III, SOA Object Systems
William M. Hegarty, ThoughtWorks
Gregor Hohpe, Google
Ralph Johnson, UIUC
James Kinneavy, University of California
Robert Laird, IBM
Doug Lea, Oswego State University of New York
Canyang Kevin Liu, SAP
Terry Lottes, Northrop Grumman Mission Systems
Chris Madrid, Microsoft
Anne Thomas Manes, Burton Group

Dragos Manolescu, Microsoft

Steven Martin, Microsoft

Joe McKendrick

J.D. Meier, Microsoft

David Michalowicz, MITRE Corporation

Per Vonge Nielsen, Microsoft

Wendell Ocasio, DoD Military Health Systems, Agilex Technologies

Philipp Offermann, University of Berlin

Dmitri Ossipov, Microsoft

Prasen Palvakar, Oracle

Parviz Peiravi, Intel

Nishit Rao, Oracle

Ian Robinson, ThoughtWorks

Richard Van Schelven, Ericsson

Shakti Sharma, Sysco Corp

Don Smith, Microsoft

Michael Sor, Booz Allen Hamilton

John Sparks, Western Southern Life

Sona Srinivasan, CISCO

Linda Terlouw, Ordina

Phil Thomas, IBM

Steve Vinoski, IEEE

Herbjörn Wilhelmsen, Objectware

Peter B. Woodhull, Modus21

Bobby Woolf, IBM

Farzin Yashar, IBM

Markus Zirn, Oracle

Olaf Zimmermann, IBM

There were many more individuals who directly or indirectly supported this effort. Amidst the flurry of correspondence over the past three years, I was unable to keep track of all participants. If you were part of the SOA design patterns project and you don't see your name on this list, then do contact me via www.thomaserl.com.

Contributors

In alphabetical order by last name:

Larry Brader

David Chappell, Oracle

Frederick Chong

Pablo Cibraro, Lagash Systems SA

Ward Cunningham

Nelly Delgado, Microsoft

Florent Georges

Charles Stacy Harris, Microsoft

Kelvin Henney, Curbralan

Jason Hogg, Microsoft

Tom Hollander

Anish Karmarkar, Oracle

Sajjad Nasir Imran, WinWire

Berthold Maier, Oracle

Hajo Normann, EDS

Wojtek Kozaczynski

Mark Little, Red Hat

Brian Lokhorst, Dutch Tax Office

Brian Loesgen, Neudesic

Matt Long, Microsoft

David Orchard, Oracle

Thomas Rischbeck, IPT

Chris Riley, SOA Systems

Satadru Roy, Sun Microsystems

Arnaud Simon, Red Hat

Paul Slater, Wadeware

Don Smith

Sharon Smith, Microsoft

Dwayne Taylor

Tina Tech

Bernd Trops, SOPER GmbH

Clemens Utschig-Utschig, Oracle

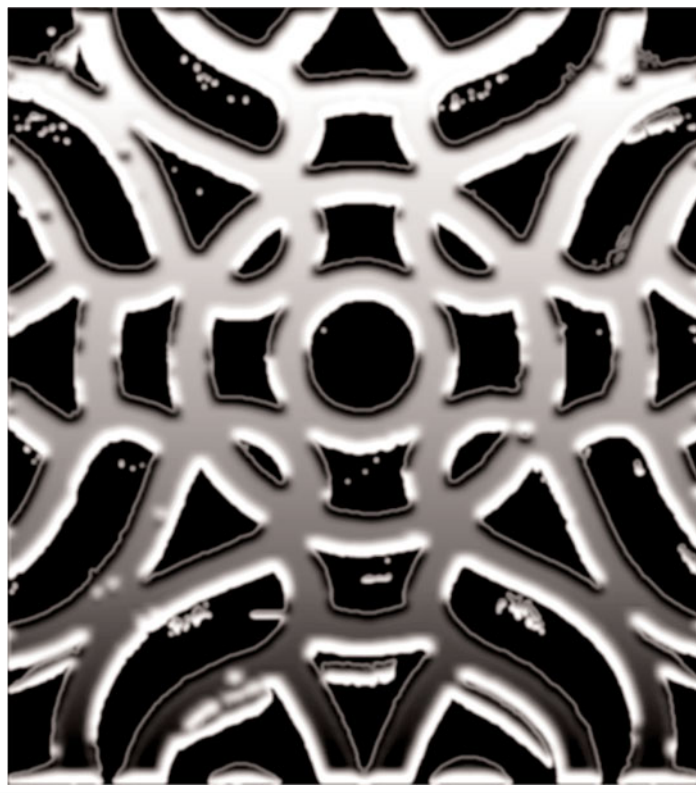
Lonnie Wall, RDA Corporation

Torsten Winterberg, Oracle

Dennis Wisnosky, U.S. Department of Defense

This page intentionally left blank

Chapter 16



Service Governance Patterns

Compatible Change

Version Identification

Termination Notification

Service Refactoring

Service Decomposition

Proxy Capability

Decomposed Capability

Distributed Capability

NOTE

The governance patterns in this chapter focus only on design-related governance issues that pertain to service architecture. The upcoming book *SOA Governance* as part of this book series will provide a collection of broader technical and organizational best practices and patterns.

Despite best efforts during analysis and modeling phases to deliver services with a broad range of capabilities, they will still be subjected to new situations and requirements that can challenge the scope of their original design. For this reason, several patterns have emerged to help evolve a service without compromising its responsibilities as an active member of a service inventory.

Compatible Change (465) and Version Identification (472) are focused on the versioning of service contracts. Similarly, Termination Notification (478) addresses the retirement of services or service contracts.

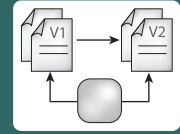
The most fundamental pattern in this chapter is Service Refactoring (484), which leverages a loosely (and ideally decoupled) contract to allow the underlying logic and implementation to be upgraded and improved.

The trio of Service Decomposition (489), Decomposed Capability (504), and Proxy Capability (497) establish techniques that allow coarser-grained services to be physically partitioned into multiple fine-grained services that can help further improve composition performance. Distributed Capability (510) also provides a specialized, refactoring-related design solution to help increase service scalability via internally distributed processing deferral.

Compatible Change

By David Orchard, Chris Riley

How can a service contract be modified without impacting consumers?



Problem	Changing an already-published service contract can impact and invalidate existing consumer programs.
Solution	Some changes to the service contract can be backwards-compatible, thereby avoiding negative consumer impacts.
Application	Service contract changes can be accommodated via extension or by the loosening of existing constraints or by applying Concurrent Contracts (421).
Impacts	Compatible changes still introduce versioning governance effort, and the technique of loosening constraints can lead to vague contract designs.
Principles	Standardized Service Contract, Service Loose Coupling
Architecture	Service

Table 16.1

Profile summary for the Compatible Change pattern.

Problem

After a service is initially deployed as part of an active service inventory, it will make its capabilities available as an enterprise resource. Consumers will be designed to invoke and interact with the service via its contract in order to leverage its capabilities for their own use. As a result, dependencies will naturally be formed between the service contract and those consumer programs. If the contract needs to be changed thereafter, that change can risk impacting existing consumers that were designed in accordance with the original, unchanged contract (Figure 16.1).

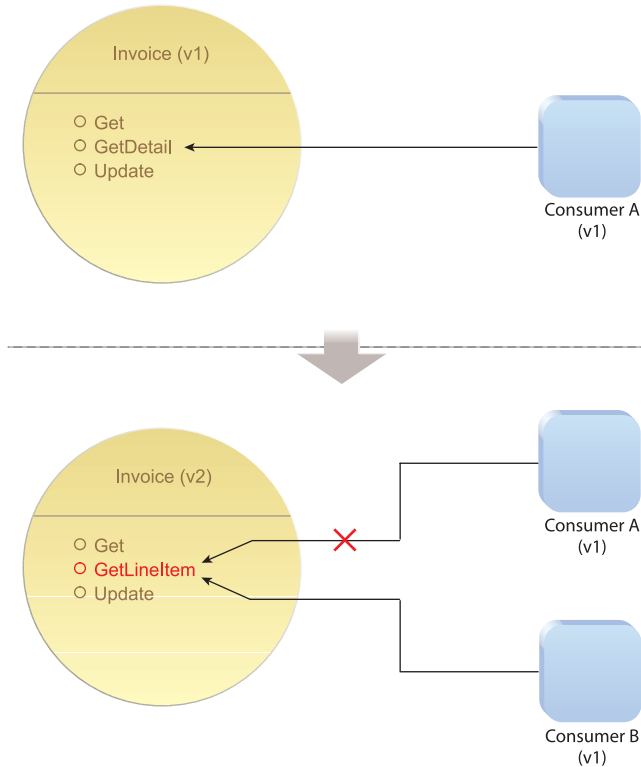


Figure 16.1
The name of a service capability is modified after version 1 of a service contract is already in use. As a result, version 2 of the contract is incompatible with Consumer A.

Solution

Wherever possible, changes to established service contracts can be made to preserve the contract’s backwards compatibility with existing consumers. This allows the service contract to evolve as required, while avoiding negative impact on dependent compositions and consumer programs (Figure 16.2).

Application

There are a variety of techniques by which this pattern can be applied, depending on the nature of the required change to the contract. The fundamental purpose of this pattern is to avoid having to impose *incompatible* changes upon a service contract that do not

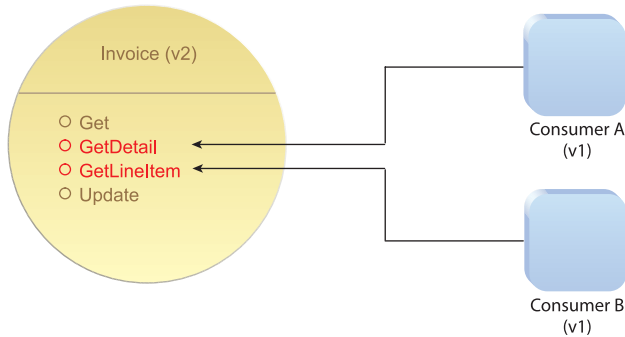


Figure 16.2

The existing capability is not renamed. Instead, a new capability with a new name is added alongside the original capability, thereby preserving compatibility with both Consumers A and B.

preserve backwards compatibility and therefore risk breaking and invalidating existing service-consumer relationships.

Here is a collection of common changes for Web service contracts, along with descriptions of how (or to what extent) these changes can be applied in a backwards-compatible manner:

- *Adding a New Operation to a WSDL Definition* – The operation can simply be appended to the existing definition, thereby acting as an extension of the contract without impacting any established contract content.
- *Renaming an Existing Operation* – As explained in the previous diagrams, an operation can be renamed by adding a new operation with the new name alongside of the existing operation with the old name. This approach can be further supplemented with Termination Notification (478), if there is a requirement to eventually retire the original operation while allowing consumers dependent on that operation a grace period to be updated in support of the renamed operation.
- *Removing an Existing Operation* – If an operation needs to be permanently deleted from the WSDL definition, there are no options for accomplishing this change in a compatible manner. Termination Notification (478) is highly recommended in this case in order to give consumer designers sufficient opportunity to transition their programs so that they are no longer using the to-be-terminated operation. Also, the technique of turning removed operations into functional stubs that respond with descriptive error data can also be employed to minimize impact on consumers that could not be transitioned.

- *Changing the MEP of an Existing Operation* – To alter an operation’s message exchange pattern requires that its input and output message definitions (and possibly its fault definition) be modified, which is normally an incompatible change. To still proceed with this change while preserving backwards compatibility requires that a new operation with the modified MEP be appended to the WSDL definition together with the original operation. As when renaming an operation in this manner, Termination Notification (478) can be used to assist an eventual transition.
- *Adding a Fault Message to an Existing Operation* – The addition of a fault message (when considered separately from a change to the MEP) may often appear as a compatible change because the option of issuing a fault message does not affect the core functionality of an operation. However, because this addition augments the service behavior, it should be considered a change that can only be compatible when adding the fault message as part of a new operation altogether.
- *Adding a New Port Type* – Because WSDL definitions allow for the existence of multiple port type definitions, the service contract can be extended by adding a new port type alongside an existing one. Although this represents a form of compatible change, it may be more desirable to simply issue a new version of the entire Web service contract.
- *Adding a New Message Schema Element or Attribute* – New elements or attributes can be added to an existing message schema as a compatible change as long as they are optional. This way, their presence will not affect established service consumers that were designed prior to their existence.
- *Removing an Existing Message Schema Element or Attribute* – Regardless of whether they are optional or required, if already established message schema elements or attributes need to be removed from the service contract, it will result in an incompatible change. Therefore, this pattern cannot be applied in this case.
- *Modifying the Constraint of an Existing Message Schema* – The validation logic behind any given part of a message schema can be modified as part of Compatible Change, as long as the constraint granularity becomes coarser. In other words, if the restrictions are loosened, then message exchanges with existing consumers should remain unaffected.
- *Adding a New Policy* – One or more WS-Policy statements can be added via Compatible Change by simply adding policy alternatives to the existing policy attachment point.

- *Adding Policy Assertions* – A policy assertion can be added as per Compatible Change (465) to an existing policy as long as it is optional or added as part of a separate policy as a policy alternative.
- *Adding Ignorable Policy Assertions* – Because ignorable policy assertions are often used to express behavioral characteristics of a service, this type of change is generally not considered compatible.

NOTE

This list of changes corresponds to a series of sections within Chapters 21, 22, and 23 in the book *Web Service Contract Design and Versioning for SOA*, which explores compatible and incompatible change scenarios with code examples.

Impacts

Each time an already published service contract is changed, versioning and governance effort is required to ensure that the change is represented as a new version of the contract and properly expressed and communicated to existing and new consumers. As explained in the upcoming *Relationships* section, this leads to a reliance upon Canonical Versioning (286) and Version Identification (472).

When applying Compatible Change in such a manner that it introduces redundancy or duplication into a contract (as explained in several of the scenarios from the *Application* section), this pattern can eventually result in bloated contracts that are more difficult to maintain. Furthermore, these techniques often lead to the need for Termination Notification (478), which can add to both the contract content and governance effort for service and consumer owners.

Finally, when the result of applying this pattern is a loosening of established contract constraints (as described in the *Modifying the Constraint of an Existing Message Schema* scenario from the *Application* section earlier), it can produce vague and overly coarse-grained contract content.

Relationships

To apply this pattern consistently across multiple services requires the presence of a formal versioning system, which is ideally standardized as per Canonical Versioning (286). Furthermore, this pattern is dependent upon Version Identification (472) to ensure that changes are properly expressed and may also require Termination Notification (478) to transition contract content and consumers from old to new versions.

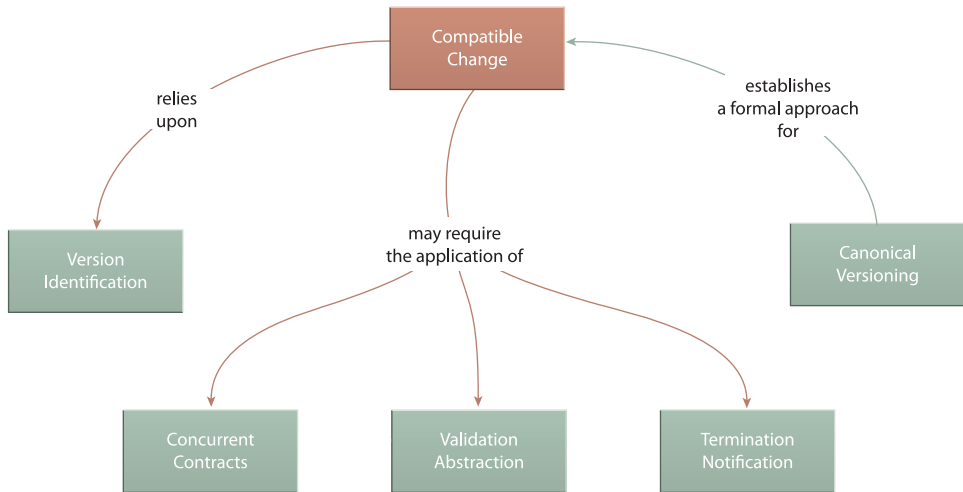


Figure 16.3

Compatible Change relates to several other service governance patterns but also may depend on some contract design patterns.

CASE STUDY EXAMPLE

As described in the case study example for Contract Denormalization (414), the Officer service contract was recently extended by FRC architects to include a new UpdateLog operation.

Before the architects can release this contract into the production environment, it must go through a testing process and be approved by the quality assurance team. The first concern raised by this team is the fact that a change has been made to the technical interface of the service and that regression testing must be carried out to ensure that existing consumers are not negatively impacted.

The architects plead with the QA manager that these additional testing cycles are not necessary. They explain that the contract content was only appended by the addition of the UpdateLog operation, and none of the previously existing contract code was affected, as shown in the highlighted parts of this example:

```

<definitions name="Officer"
  targetNamespace="http://frc/officer/wsdl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:off="http://frc/officer/schema/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

```

```

xmlns:tns="http://frc/officer/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
  <xsd:schema targetNamespace="http://frc/officer/">
    <xsd:import namespace="http://frc/officer/schema/"
      schemaLocation="Officer.xsd"/>
  </xsd:schema>
</types>
<message name="UpdateOfficer">
  <part name="RequestA" element="off:OfficerDoc" />
</message>
<message name="UpdateOfficerConfirm">
  <part name="ResponseA" element="off:ReturnCodeA" />
</message>
<message name="UpdateOfficerLog">
  <part name="RequestB" element="off:OfficerLog" />
</message>
<message name="UpdateOfficerLogConfirm">
  <part name="ResponseB" element="off:ReturnCodeB" />
</message>
<portType name="OffInt">
  <operation name="Update">
    <input message="tns:UpdateOfficer" />
    <output message="tns:UpdateOfficerConfirm" />
  </operation>
  <operation name="UpdateLog">
    <input message="tns:UpdateOfficerLog" />
    <output message="tns:UpdateOfficerLogConfirm" />
  </operation>
</portType>
  ...
</definitions>

```

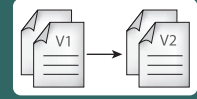
Example 16.1

The WSDL definition from Example 14.1 is revisited to show how the change made during the application of Contract Denormalization (414) was backwards-compatible.

Because existing content was not changed and only new content was added, they claim that the contract is fully backwards-compatible. The QA manager agrees that this indicates a reduced risk but insists that the revised service be subjected to some testing to ensure that the addition of the new operation logic did not affect its overall behavior.

Version Identification

By David Orchard, Chris Riley



How can consumers be made aware of service contract version information?

Problem	When an already-published service contract is changed, unaware consumers will miss the opportunity to leverage the change or may be negatively impacted by the change.
Solution	Versioning information pertaining to compatible and incompatible changes can be expressed as part of the service contract, both for communication and enforcement purposes.
Application	With Web service contracts, version numbers can be incorporated into namespace values and as annotations.
Impacts	This pattern may require that version information be expressed with a proprietary vocabulary that needs to be understood by consumer designers in advance.
Principles	Standardized Service Contract
Architecture	Service

Table 16.2

Profile summary for the Version Identification pattern.

Problem

Whether a contract is subject to compatible or incompatible changes, any modification to its published content will typically warrant a new contract version. Without a means of associating contract versions with changes, the compatibility between a service and its current and new consumers is constantly at risk, and the service also becomes less discoverable to consumer designers (Figure 16.4).

Furthermore, the service itself also becomes more burdensome to govern and evolve.

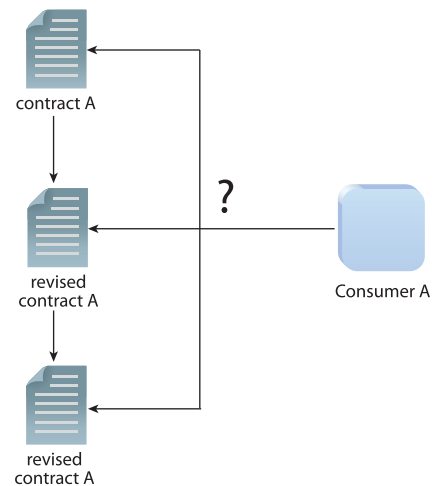


Figure 16.4

As a service contract is required to change, a service consumer is left in the dark as to whether it is still compatible.

Solution

The service contract can be designed to express version identifiers that allow the consumer to confidently determine whether it is compatible with the service. The use of version identifiers further supports Concurrent Contracts (421) for versioning purposes, thereby allowing a consumer to choose the correct contract based on its expressed version, as shown in Figure 16.5.

Application

Versions are typically identified using numeric values that are incorporated into the service contract either as human-readable annotations or as actual extensions of the technical contract content. The most common version number format is a decimal where the first digit represents the major version number, and digits following the decimal point represent minor version numbers.

What the version numbers actually mean depends on the conventions established by an overarching versioning strategy. Two common approaches are described here:

- *Amount of Work* – Major and minor version numbers are used to indicate the amount of effort that went into each change. An increment of a major version number represents a significant amount of work, whereas increases in the minor version numbers represent minor upgrades.
- *Compatibility Guarantee* – Major and minor version numbers are used to express compatibility. The most common system is based on the rule that an increase in a major version number will result in a contract that is not backwards-compatible, whereas increases in minor version numbers are backwards-compatible. As a result, minor version increments are not expected to affect existing consumers.

Note that these two identification systems can be combined so that version number increases continue to indicate compatible or incompatible changes, while also representing the amount of work that went into the changes.

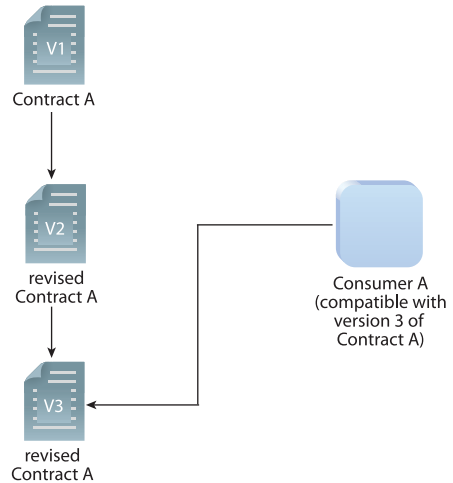


Figure 16.5

Because the service contracts express versioning information, Consumer A can proceed to invoke version 3 of the service contract because it was designed to be compatible with that specific version.

With Web service contracts specifically, a common means of ensuring that existing consumers cannot accidentally bind to contracts that have been subject to non-backwards-compatible changes is to incorporate the version numbers into new namespace values that are issued with each new major version increase.

NOTE

Whereas version numbers are often incorporated into the target namespaces for WSDL definitions, date values are commonly appended to target namespaces for XML Schema definitions. See Chapters 20, 21, and 22 in the *Web Service Contract Design and Versioning for SOA* book for code examples and more details.

Impacts

Version identification systems and conventions are typically specific to a given service inventory and usually part of a standardized versioning strategy, as per Canonical Versioning (286). As a result, they are not standardized on an industry level and therefore, when expressed as part of the technical contract, impose the constant requirement that service consumers be designed to understand the meaning of version identifiers and programmatically consume them, as required.

When services are exposed to new or external consumers, these same requirements apply, but the necessary enforcement of standards may be more difficult to achieve.

Relationships

This pattern is commonly applied together with (or as a result of) the application of Canonical Versioning (286) and is further an essential part of carrying out Compatible Change (465).

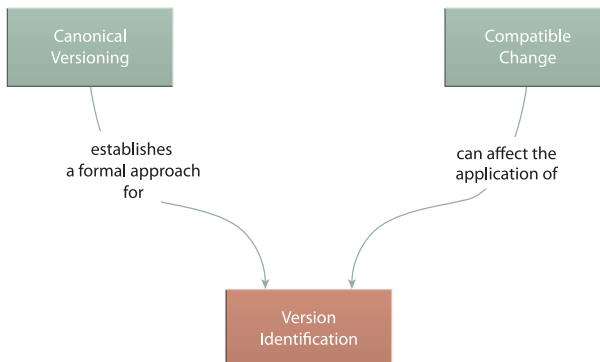


Figure 16.6

Version Identification relates primarily to other contract versioning patterns.

NOTE

Version Identification is comparable to Format Indicator (Hohpe, Woolf) when applied to express a version number as part of a message. Format Indicator differs in that it is message-centric and also enables the expression of other meta information, such as foreign keys and document formats.

CASE STUDY EXAMPLE

As explained in the example for Compatible Change (465) example, the quality assurance team performs some further testing on the Officer service with its extended service contract. Subsequent to carrying out these tests they clear the new service for release into production, subject to one condition: The service contract must express a version number to indicate that it has been changed.

This version number follows existing versioning conventions whereby a backwards-compatible change increments the minor version number (the digit following the decimal point). The FRC architects agree that this is a good idea and are quick to add a human-readable comment to the Officer WSDL definition by using the `documentation` element as follows:

```
<definitions name="Officer"
  targetNamespace="http://frc/officer/wsd1/"
  xmlns="http://schemas.xmlsoap.org/wsd1/"
  xmlns:off="http://frc/officer/schema/"
  xmlns:soap="http://schemas.xmlsoap.org/wsd1/soap/"
  xmlns:tns="http://frc/officer/wsd1/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <documentation>Version 1.1</documentation>
  ...
</definitions>
```

Example 16.2

The WSDL definition from Example 16.1 is annotated with a version number.

Several months after version 1.1 of the Officer service was deployed, two new projects get started, each pertaining to the HR system that is currently in place. One of the FRC enterprise architects is involved with both project teams to ensure compliance with design standards. After reviewing each design specification, she notices some commonality. The first solution requires event logging functionality, and the other solution has a requirement for error logging. She soon realizes that there is a need for the FRC to create a separate utility Logging service.

After proposing her recommendation, the Logging service is built in support of both solutions. Weeks later during a review of the service inventory blueprint, an analyst points out that the UpdateLog operation that was added to the Officer service contract should, in fact, be located within the new Logging service.

The FRC architecture team agrees to make this change, though it isn't considered an immediate priority. Several weeks thereafter, the Officer service is revisited, and the logic behind the UpdateLog operation is removed. As a result, the UpdateLog operation itself is deleted from the contract.

Following the versioning conventions set out by the quality assurance team, this type of change is classified as "incompatible," meaning that it imposes a non-backwards-compatible change that will impact consumer programs that have already formed dependencies on the Officer service's UpdateLog operation. Consequently, they are required to increment the major version number (the digit before the decimal) and further append the Officer WSDL definition's target namespace with the new version number, as follows:

```
<definitions name="Officer"
  targetNamespace="http://frc/officer/wSDL/v2"
  xmlns="http://schemas.xmlsoap.org/wSDL/"
  xmlns:off="http://frc/officer/schema/"
  xmlns:soap="http://schemas.xmlsoap.org/wSDL/soap/"
  xmlns:tns="http://frc/officer/wSDL/v2"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <documentation>Version 2.0</documentation>
  <types>
    <xsd:schema targetNamespace="http://frc/officer/">
      <xsd:import namespace="http://frc/officer/schema/"
        schemaLocation="Officer.xsd"/>
    </xsd:schema>
  </types>
  <message name="UpdateOfficer">
    <part name="RequestA" element="off:OfficerDoc"/>
  </message>
  <message name="UpdateOfficerConfirm">
    <part name="ResponseA" element="off:ReturnCodeA"/>
  </message>
  <portType name="OffInt">
    <operation name="Update">
      <input message="tns:UpdateOfficer"/>
      <output message="tns:UpdateOfficerConfirm"/>
    </operation>
  </portType>
</definitions>
```

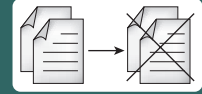
```
</portType>  
...  
</definitions>
```

Example 16.3

The UpdateLog operation is removed from the revised Officer WSDL definition, resulting in an incompatible change that requires a new target namespace value.

Termination Notification

By David Orchard, Chris Riley



How can the scheduled expiry of a service contract be communicated to consumer programs?

Problem	Consumer programs may be unaware of when a service or a service contract version is scheduled for retirement, thereby risking runtime failure.
Solution	Service contracts can be designed to express termination information for programmatic and human consumption.
Application	Service contracts can be extended with ignorable policy assertions or supplemented with human-readable annotations.
Impacts	The syntax and conventions used to express termination information must be understood by service consumers in order for this information to be effectively used.
Principles	Standardized Service Contract
Architecture	Composition, Service

Table 16.3

Profile summary for the Termination Notification pattern.

Problem

As services evolve over time, various conditions and circumstances can lead to the need to retire a service contract, a portion of a service contract, or the entire service itself.

Examples include:

- the service contract is subjected to a non-backwards-compatible change
- a compatible change is applied to a service contract but strict versioning policies require the issuance of an entirely new version of the service contract
- a service's original functional scope is no longer applicable in relation to how the business has changed
- a service is decomposed into more granular services or combined together with another service

In larger IT enterprises and especially when making services accessible to external partner organizations, it can be challenging to communicate to consumer owners the pending termination of a service or any part of its contract in a timely manner.

Failure to recognize a scheduled retirement will inevitably lead to runtime failure scenarios, where unaware consumer programs that attempt to invoke the service are rejected (Figure 16.7).

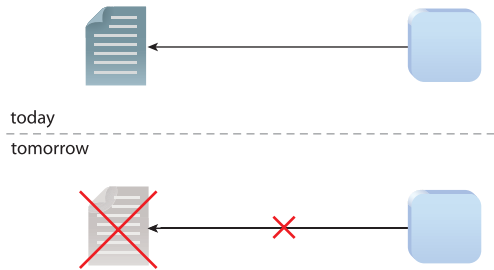


Figure 16.7

The consumer program (right) invokes a service via its contract as usual today, but when the contract is terminated on the next day, the attempted invocation fails.

Solution

Service contracts are equipped with termination details, thereby allowing consumers to become aware of the contract retirement in advance (Figure 16.8).

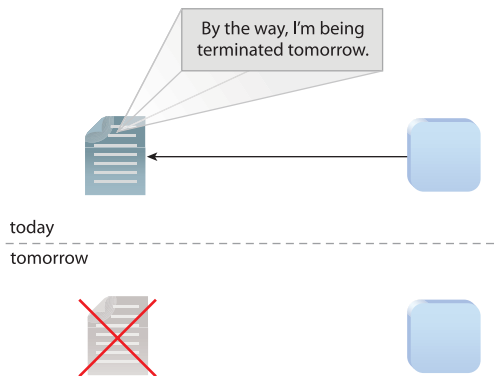


Figure 16.8

The service contract includes a standardized statement that communicates when it is scheduled for termination. As a result, the consumer does not attempt to invoke it after the contract has been terminated.

Application

This pattern is most commonly applied by supplementing technical contract content with human-readable annotations that simply provide the termination date. However, with Web service contracts, there is also the option of leveraging the WS-Policy language to express termination notifications via ignorable policy assertions. This enables consumer programs to be designed to programmatically check for termination information.

It is also worth noting that in addition to expressing service contract termination, there are other purposes for which Termination Notification can be applied, such as:

- *Indicating the retirement of a specific capability or operation* – This is especially relevant when choosing one of the transition techniques described in Compatible Change (465) where an original operation is preserved and a similar, but changed, operation is added.
- *Indicating the retirement of an entire service* – This same approach can be used to communicate that an entire service program itself is scheduled for retirement.
- *Indicating the retirement of a message schema* – Although policy assertions may not be suitable for this purpose, regular annotations can be added to schemas to explain when the schema version will be terminated and/or replaced.

Note also that governance standards can be put in place as part of an overarching Canonical Versioning (286) strategy to express termination notification information via standardized annotations or non-ignorable policy assertions. In the latter case, this can require that all Web service contracts contain termination assertions, regardless of whether they are due for termination. For those contracts that are not being terminated, a pre-defined value indicating this is placed in the assertion instead of a date (or the assertion is left empty).

Impacts

All of the techniques explained in this pattern description require the use of non-standardized extension content for service contracts. This is because there is no industry standard for expressing termination information. Termination Notification relies on the existence and successful enforcement of governance standards and therefore has a direct dependency on Canonical Versioning (286).

Relationships

As just mentioned, how this pattern is applied is often governed by Canonical Versioning (286). Both Compatible Change (465) and Proxy Capability (497) can lead to the need for Termination Notification.

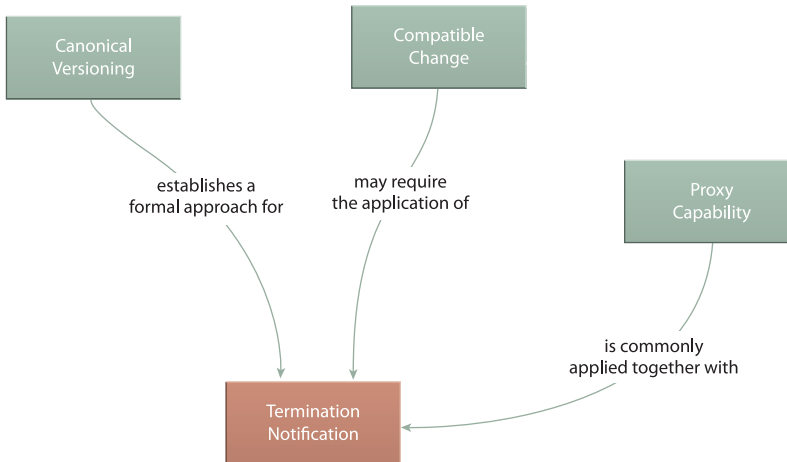


Figure 16.9

Termination Notification relates primarily to other versioning patterns but also can support Proxy Capability (497).

NOTE

Termination Notification is similar in concept to Message Expiration (Hohpe, Woolf), a pattern that advocates adding a timestamp to a message to indicate when the message itself is no longer considered valid.

CASE STUDY EXAMPLE

In the example for Version Identification (472) example the FRC team removed the UpdateLog operation from the Officer WSDL definition, resulting in a non-backwards-compatible change. After a meeting with some of the custodians of the consumer programs affected by this incompatible change, FRC architects begin to realize that the change will result in significant effort on the part of consumer owners and that it could take several months before all of the consumer programs are updated to work with the new utility Logging service. Furthermore, several of the individuals responsible for owning consumers were not available and will need to be informed of the pending change at a later point.

As a result of these circumstances, the FRC team decides to postpone the change, allowing the Officer service to maintain its UpdateLog operation for the next six months while the Logging service is also available. They work with the quality assurance team on a plan to accommodate this transition, as follows:

1. Establish a new design standard that disallows any new consumer programs from accessing the Officer service's UpdateLog operation.
2. Notify all team leads via e-mail of the date on which the UpdateLog operation will be removed.
3. Incorporate this termination date into the Officer WSDL definition by means of a non-ignorable policy assertion.

Step 3 is implemented as follows:

```
<definitions name="Officer" ... >
...
<binding name="bdPO" type="tns:OffInt">
  <operation name="Update">
    <soapbind:operation
      soapAction="http://frc/update/request"
      soapActionRequired="true" required="true"/>
    <input>
      <soapbind:body use="literal"/>
    </input>
    <output>
      <soapbind:body use="literal"/>
    </output>
  </operation>
  <operation name="UpdateLog">
    <wsp:Policy>
      <pol:termination wsp:Ignorable="true">
        Mar-01-2009
      </pol:termination>
    </wsp:Policy>
    <soapbind:operation
      soapAction="http://frc/updateLog/request"
      soapActionRequired="true" required="true"/>
    <input>
      <soapbind:body use="literal"/>
    </input>
    <output>
      <soapbind:body use="literal"/>
    </output>
  </operation>
</binding>
</definitions>
```

```
    </output>
  </operation>
  ...
</binding>
...
</definitions>
```

Example 16.4

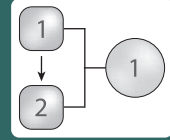
The `operation` element within the Officer WSDL definition's `binding` construct is modified to include a custom ignorable WS-Policy assertion that expresses the scheduled termination date of the UpdateLog operation.

NOTE

For more examples of Termination Notification see Chapter 23 of *Web Service Contract Design and Versioning for SOA*.

Service Refactoring

How can a service be evolved without impacting existing consumers?



Problem	The logic or implementation technology of a service may become outdated or inadequate over time, but the service has become too entrenched to be replaced.
Solution	The service contract is preserved to maintain existing consumer dependencies, but the underlying service logic and/or implementation are refactored.
Application	Service logic and implementation technology are gradually improved or upgraded but must undergo additional testing.
Impacts	This pattern introduces governance effort as well as risk associated with potentially negative side-effects introduced by new logic or technology.
Principles	Standardized Service Contract, Service Loose Coupling, Service Abstraction
Architecture	Service

Table 16.4
Profile summary for the Service Refactoring pattern.

Problem

Subsequent to its initial delivery, unforeseen performance and business requirements may demand more from a service than it is capable of providing (Figure 16.10). Replacing the service entirely may be undesirable, especially when several consumer programs have already formed dependencies upon its established service contract.

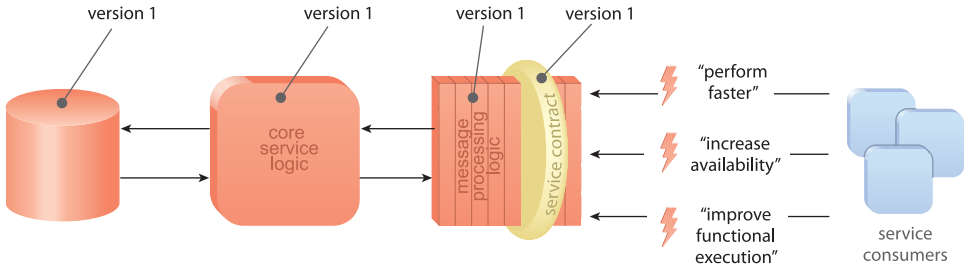


Figure 16.10

Consumers of an existing service demand new requirements for which the service was not originally designed. The red symbols indicate the different parts of this service architecture that could be independently versioned.

Solution

Software refactoring is an accepted software engineering practice whereby existing software programs can be gradually improved without affecting the manner in which they behave. When applied to service design, this approach provides more opportunity for services to evolve within an organization without disrupting their existing consumers. As shown in Figure 16.11, with the application of this pattern the underlying logic and implementation of a service can be regularly optimized, improved, or even upgraded while preserving the service contract.

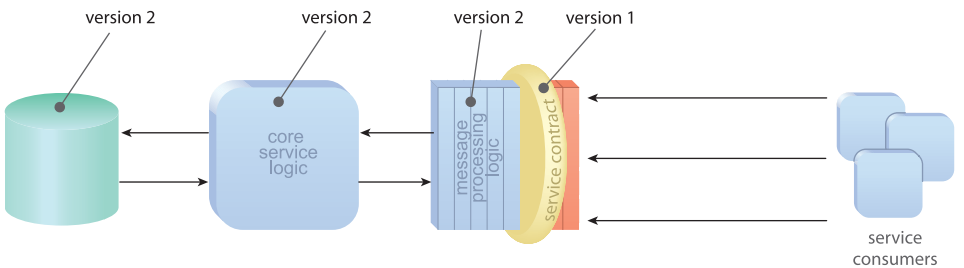


Figure 16.11

All parts of a service architecture abstracted by its contract can potentially be refactored without compromising existing consumer relationships. The service contract and the remaining, externally facing message processing agents (red) are not affected by the refactoring effort.

Application

Software refactoring practices allow programs to be improved through a series of small upgrades that continue to preserve their interfaces and overall behavior. By limiting the scope of these upgrades, the risk associated with negatively impacting consumers is minimized. The emphasis of software refactoring techniques is on the cumulative result of these individual refactoring steps.

This pattern can be more successfully applied when the service has already been subjected to the application of Decoupled Contract (401) and the Service Loose Coupling design principle. The separation of service logic from a fully decoupled contract provides increased freedom as to how refactoring can be carried out, while minimizing potential disruption to existing service consumers.

NOTE

Several books covering refactoring techniques and specialized patterns are available. Two well-known titles are *Refactoring: Improving the Design of Existing Code* (Fowler, Beck, Brant, Opdyke, Roberts) and *Refactoring to Patterns* (Kerievsky), both by Addison-Wesley. The site www.refactoring.com provides additional resources as well as a catalog of proven “refactorings.”

Impacts

The refactoring of existing service logic or technology introduces the need for the service to undergo redesign, redevelopment, and retesting cycles so as to ensure that the existing guarantees expressed in the service contract (which includes its SOA) can continue to be fulfilled as expected (or better).

Because already established and proven logic and technology is modified or replaced as a result of applying this pattern, there is still always a risk that the behavior and reliability of a refactored capability or service may still somehow negatively affect existing consumers. The degree to which this risk is alleviated is proportional to the maturity, suitability, and scope of the newly added logic and technology and the extent to which quality assurance and testing are applied to the refactored service.

Relationships

The extent to which Service Refactoring can be applied depends on how the service itself was first designed. This is why there is a direct relationship between this pattern and Service Normalization (131), Contract Centralization (409), and Decoupled Contract (401). The abstraction and independence gained by the successful application of those patterns allows services to be individually governed and evolved with minimal impact to consumer programs.

Furthermore, depending on the nature of the refactoring requirements, Service Decomposition (489), Concurrent Contracts (421), or Service Façade (333) may need to be applied to accommodate how the service is being improved.

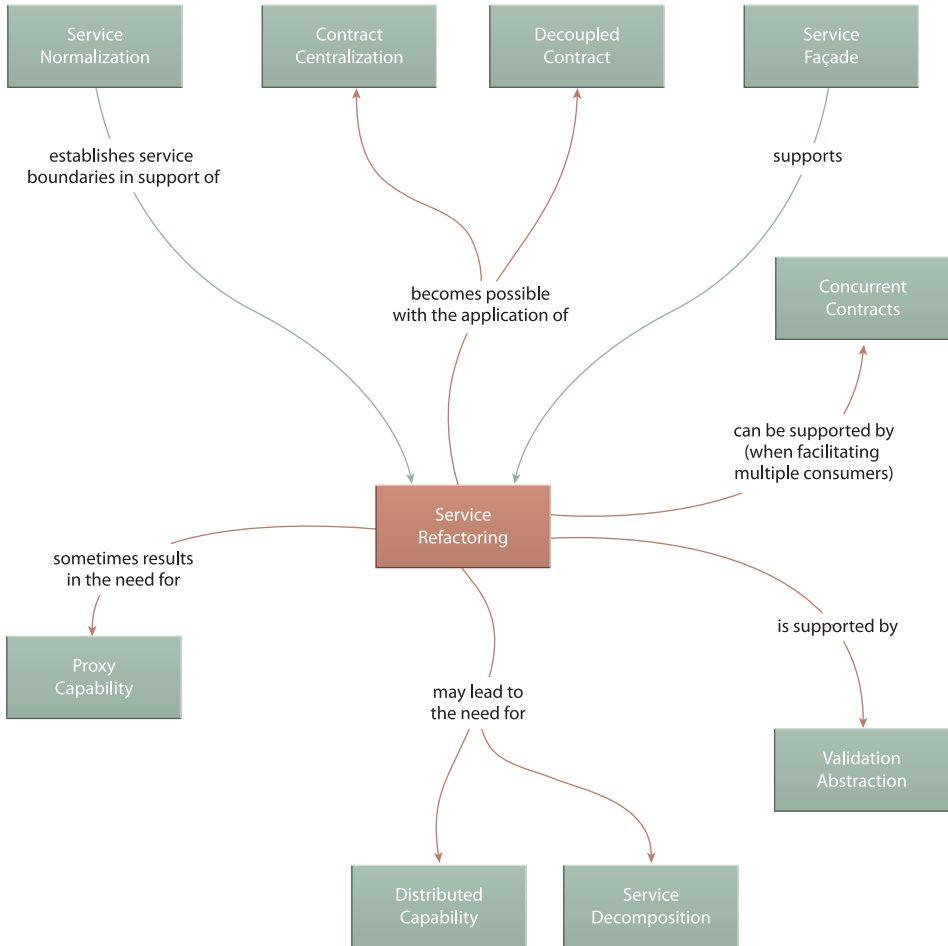


Figure 16.12

Service Refactoring relies on several key contract-related patterns to ensure that refactoring-related changes do not disrupt existing service consumers.

CASE STUDY EXAMPLE

The Alleywood Employee service was implemented some time ago. It originally established a standardized service contract that acted as an endpoint into the HR module of a large ERP system. Since the McPherson buyout, various products have been upgraded or replaced to contemporize the overall IT enterprise. As part of this initiative, this ERP system was re-evaluated.

The ERP vendor had been bought out by a competing software manufacturer, and the ERP platform was simply made part of a larger product line that offered an alternative ERP. The McPherson group believed that the original Alleywood ERP environment would soon be retired by its new owner in order to give their ERP product a greater market share.

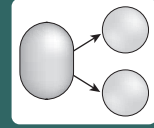
As a result, it was decided to completely replace this product. This, of course, affected many services, including the Employee service. However, because its contract was decoupled and had been fully standardized, it was in no way dependent on any part of the underlying ERP environment.

A new HR product and a custom-developed employee reporting application were introduced, allowing developers to refactor some of the core service logic so that the concurrent usage thresholds of the more popular service capabilities could be increased while the service contract and the service's overall expected behavior are preserved.

This limited the impact of the HR product to the service only. Besides a brief period of unavailability, all Employee service consumers were shielded from this impact and continued to use the Employee service as normal.

Service Decomposition

How can the granularity of a service be increased subsequent to its implementation?



Problem	Overly coarse-grained services can inhibit optimal composition design.
Solution	An already implemented coarse-grained service can be decomposed into two or more fine-grained services.
Application	The underlying service logic is restructured, and new service contracts are established. This pattern will likely require Proxy Capability (497) to preserve the integrity of the original coarse-grained service contract.
Impacts	An increase in fine-grained services naturally leads to larger, more complex service composition designs.
Principles	Service Loose Coupling, Service Composability
Architecture	Service

Table 16.5

Profile summary for the Service Decomposition pattern.

Problem

When modeling services during the initial analysis phases it is common to take practical considerations into account. For example, what may ideally be represented by a set of fine-grained business service candidates is later combined into a smaller number of coarse-grained services primarily due to performance and other infrastructure-related concerns motivated by the need to keep service composition sizes under control.

After a service inventory architecture matures and more powerful and sophisticated technology and runtime products are incorporated, larger, more complex service compositions become a reality. When designing such compositions, it is generally preferable to keep the footprints of individual services as small as possible because only select service capabilities are required to automate a given parent business task. However, when forced to work with overly coarse-grained services, composition performance can be negatively affected, and the overall composition designs can be less than optimal (Figure 16.13).

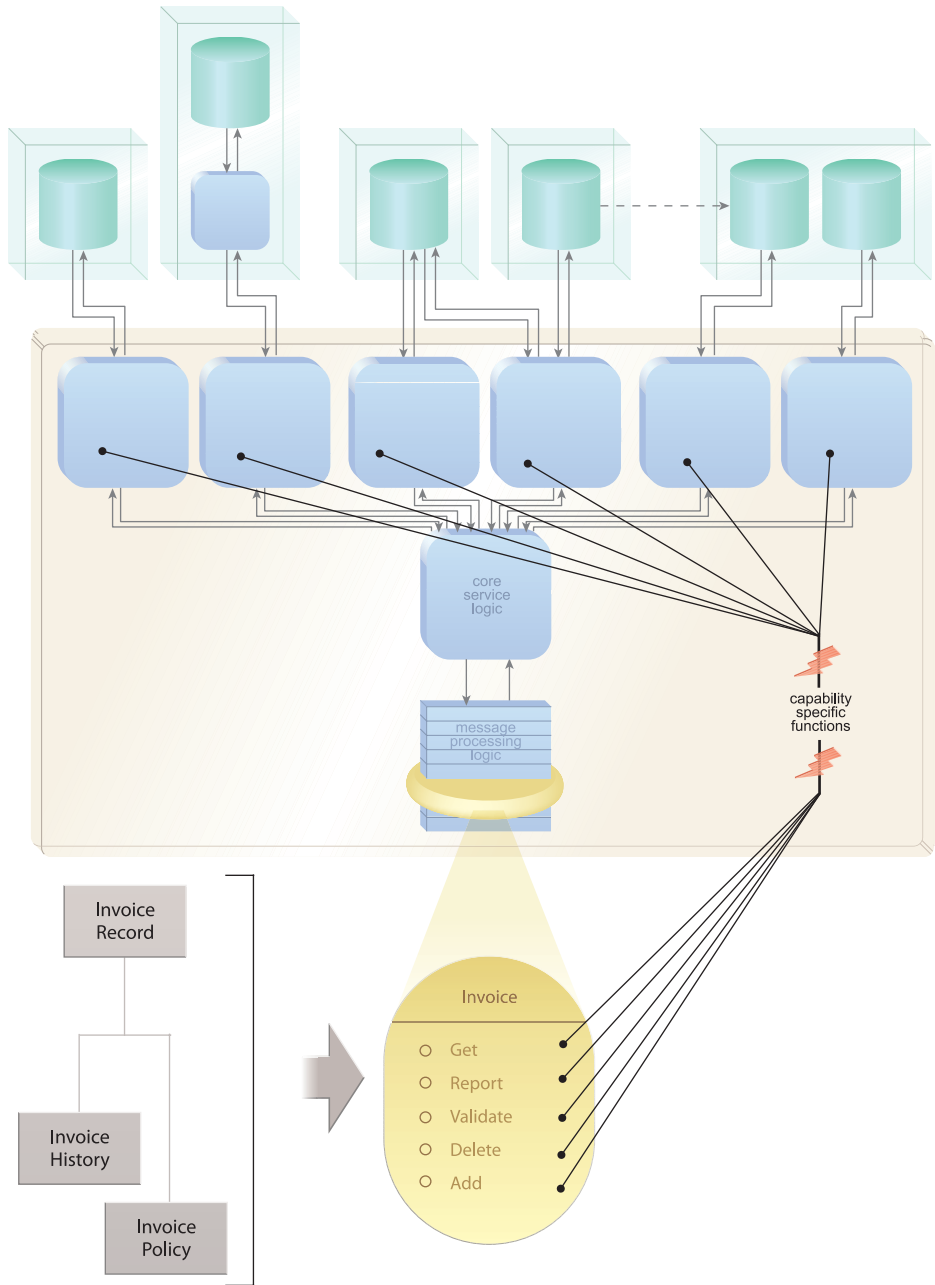


Figure 16.13

An Invoice service with a functional context originally derived from three separate business entities ends up existing as a large software program with a correspondingly large footprint, regardless of which capability a composition may need to compose.

NOTE

Another circumstance under which this problem condition can occur is when services are being produced via a meet-in-the-middle delivery process, where a top-down analysis is only partially completed prior to service development. In this delivery approach, the top-down process continues concurrently with service delivery projects. There is a commitment to revising implemented service designs after the top-down analysis progresses to a point where necessary changes to the original service inventory are identified. For more details regarding SOA project delivery strategies, see Chapter 10 in *Service-Oriented Architecture: Concepts, Technology, and Design*.

Solution

The coarse-grained service is decomposed into a set of fine-grained services that collectively represent the functional context of the original service but establish distinct functional contexts of their own (Figure 16.14).

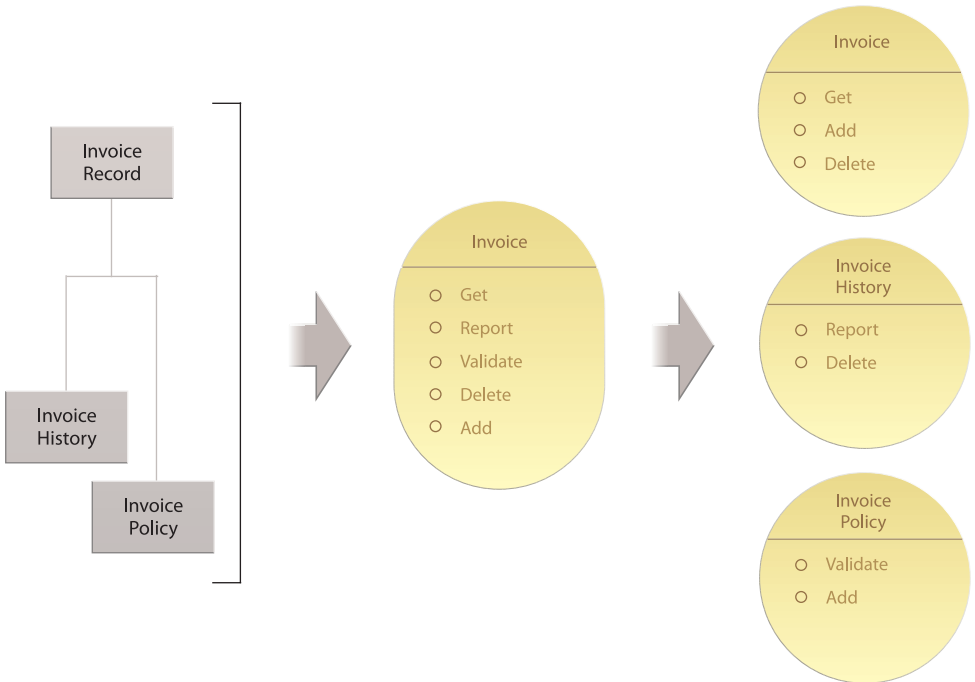


Figure 16.14

The original, coarse-grained Invoice service is decomposed into three separate services, one of which remains associated with general invoice processing but only encapsulates a subset of the original capabilities.

Application

Carrying out this pattern essentially requires that the existing, coarse-grained service be broken apart and its logic reorganized into new, finer-grained functional boundaries.

Therefore, the first step is usually to revisit the service inventory blueprint and decide how the service can be re-modeled into multiple service candidates. As part of this process, new capability candidates will also need to be defined, especially if Decomposed Capability (504) was not taken into account during the service's original design. After the modeling is completed, the new services are subject to the standard lifecycle phases, beginning with contract design (based on the modeled service candidates) and all the way through to final testing and quality assurance phases (Figure 16.15).

Unless it is decided to also retrofit previous consumer programs that formed dependencies on the original service, Proxy Capability (497) will likely need to be applied to preserve the original service contract for backwards compatibility.

NOTE

The concepts behind this pattern can also be applied in reverse, where two or more fine-grained services are combined into one coarse-grained service. The use of Proxy Capability (497) would still apply for preserving the original service contracts. This is the basis of a pattern called Service Consolidation which, at the time of this writing, was classified as a candidate pattern that is available for review at SOAPatterns.org.

Impacts

The extent to which Service Decomposition can impact a service inventory depends on how established a service is and how many consumer programs have formed relationships on it. The more consumers involved, the more complicated and disruptive this pattern can be.

Because this pattern is commonly applied after an inventory architecture has matured, its application needs to be carefully planned together with the repeated application of Proxy Capability (497).

The preventative use of Decomposed Capability (504) can ease the impact of Service Decomposition and will also result in a cleaner separation of functional service contexts.

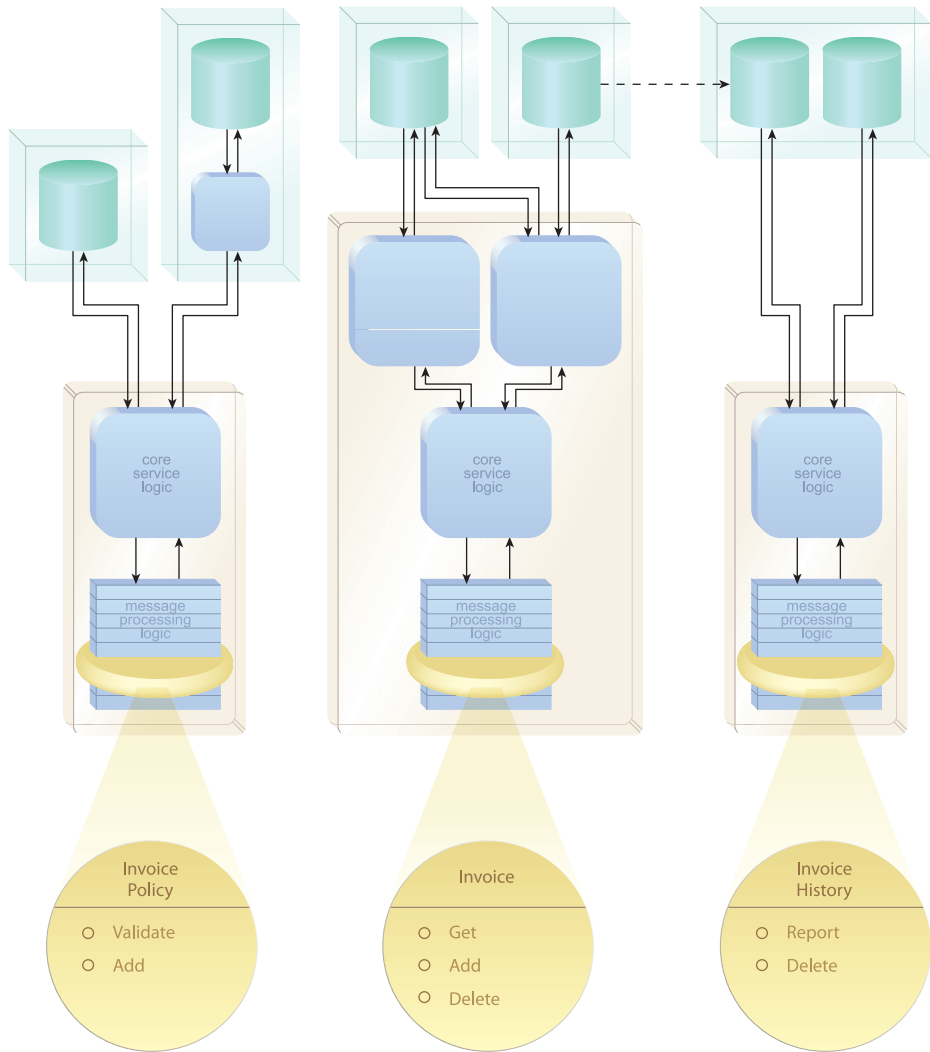


Figure 16.15

The new, fine-grained services each provide fewer capabilities and therefore also impose smaller program sizes.

Relationships

Service Decomposition has a series of relationships with other service-level patterns, most notably Service Refactoring (484). When a service is upgraded as a result of a refactoring effort, the application of Service Decomposition may very well be the means by which this is carried out.

As explained in the pattern description for Proxy Capability (497), Service Decomposition relies on that pattern to implement the actual partitioning via the redevelopment effort required to turn one or more regular capabilities into proxies. As a result, this pattern shares several of the same patterns as Proxy Capability (497).

Service Decomposition is most frequently applied to agnostic services, therefore tying it to Entity Abstraction (175) and Utility Abstraction (168). However, the result of this pattern can introduce a measure of service redundancy due to the need for Proxy Capability (497) to violate Service Normalization (131) to some extent.

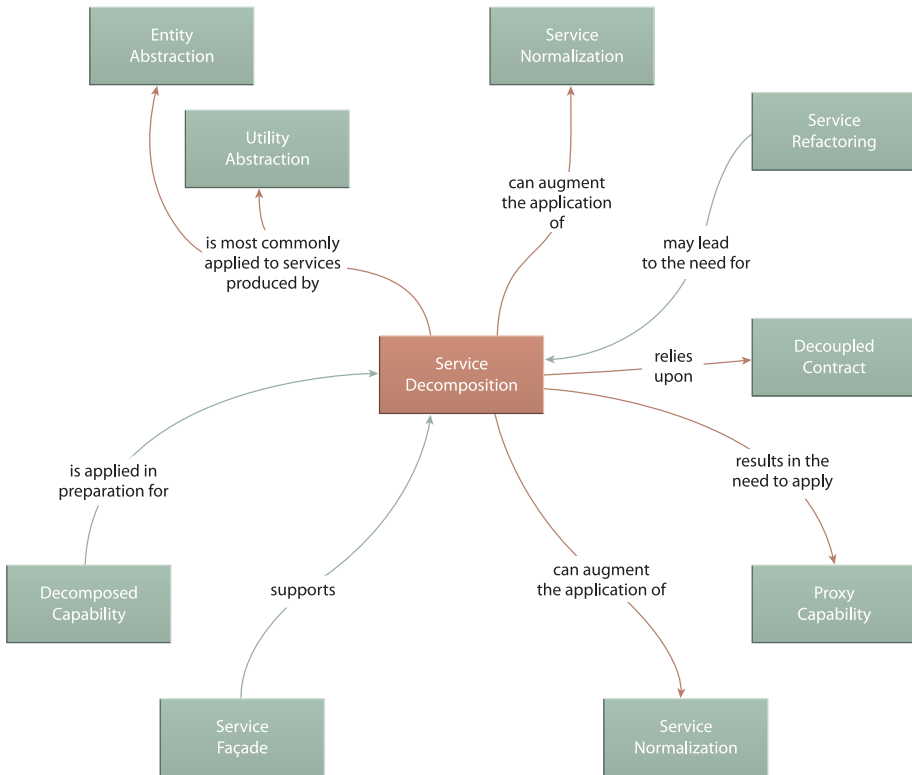


Figure 16.16 Service Decomposition is a refactoring-related approach to splitting up service logic that ties into numerous patterns that shape service logic and contracts.

CASE STUDY EXAMPLE

The case study example for Service Refactoring (484) explained how the Employee service was reengineered for a number of reasons. One of the results of this effort is that the service is now more scalable and can handle increased usage loads. The primary reason scalability was addressed is in preparation for new, upcoming service compositions that will require access to employee data and functionality. Those compositions were in the planning stages at that time and are now in production.

Some preliminary statistics show that despite the increase in usage thresholds, the Employee service is still excessively strained, and there have already been complaints about latency and memory overhead associated with the service's invocation and participation as part of the overall composition.

At first the team responsible for the Employee service considers Redundant Implementation (345) to help alleviate this situation. While this would address some of the latency issues, it would not solve the memory overhead issue.

The team then explores the option of splitting the functionality in the Employee service into two separate services. From a back-end perspective, there is an opportunity to do this in a relatively clean-cut manner. Currently, the service encapsulates functionality from an HR ERP system and a custom-developed reporting application. However, as a member of the entity service layer, the architects and business analysts involved would like to preserve the business entity-based functional context in each of the two services it would be split into. Therefore, they don't want to make the decision based on the current service implementation architecture alone.

They turn to the information architecture group responsible for maintaining the master entity relationship diagram to look for suitable employee-related entities that might form the basis of separate services. They locate an Employee Records entity that has a relationship with the parent Employee entity. Employee Records represents historical employee information, such as overtime, sick days, complaints, promotions, injuries, etc.

The team reviews the current entity service functionality and additional capabilities that may need to be added (such as those modeled as part of the service inventory blueprint but not yet implemented). They also look into the back-end systems being encapsulated. The custom-developed reporting application does not provide all of the required features to support a service dedicated to Employee Records processing. The team would need for this service to continue accessing the HR ERP system, plus

eventually upcoming Employee Records capabilities will need to further access the central data warehouse.

On the bright side, their original usage statistics indicate that some of the latency issues resulted from the Employee service being tied up executing long-running reporting queries. If this type of functionality were to exist in a separate service, the primary Employee capabilities would be more scalable and reliable, and the Employee service would be “lighter” and a more effective composition participant.

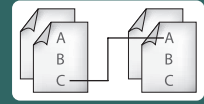
After taking all these factors into consideration, the team feels that it makes sense to break off historical reporting functionality into a separate service appropriately called “Employee Records.” The first challenge they face is that the existing Employee service contract is already being used by many consumer programs. If they move capabilities from this service to another, they will introduce significant disruption. For this situation, they apply Proxy Capability (497), as explained in the next case study example.

NOTE

The preceding scenario describes one possible option as to how a service can be decomposed. Another design option is to split the one entity service into an entity and utility service in order to accommodate more practical concerns. Either way, how a service is decomposed is ultimately best determined by a thorough analysis to ensure that your business requirements are fully met.

Proxy Capability

How can a service subject to decomposition continue to support consumers affected by the decomposition?



Problem	If an established service needs to be decomposed into multiple services, its contract and its existing consumers can be impacted.
Solution	The original service contract is preserved, even if underlying capability logic is separated, by turning the established capability definition into a proxy.
Application	Façade logic needs to be introduced to relay requests and responses between the proxy and newly located capabilities.
Impacts	The practical solution provided by this pattern results in a measure of service denormalization.
Principles	Service Loose Coupling
Architecture	Service

Table 16.6

Profile summary for the Proxy Capability pattern.

Problem

As per Service Decomposition (489), it is sometimes deemed necessary to further decompose a service's functional boundary into two or more functional boundaries, essentially establishing new services within the overall inventory. This can clearly impact existing service consumers who have already formed dependencies on the established service contract (Figure 16.17).

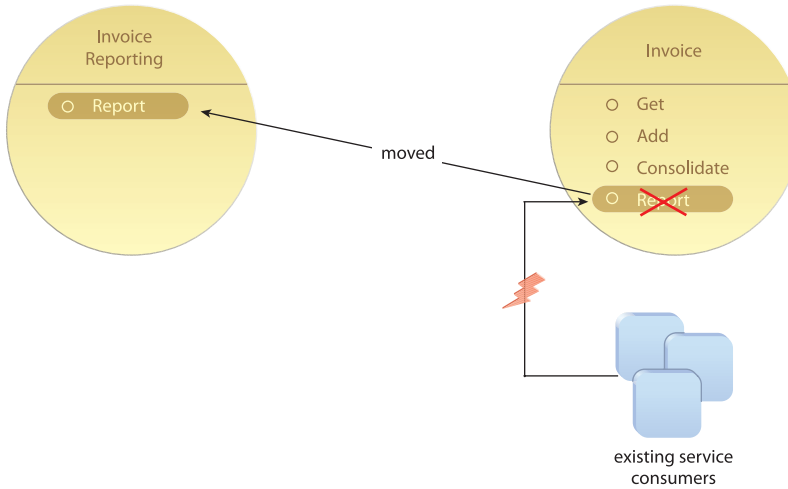


Figure 16.17

Moving a service capability that is part of an established service contract will predictably impact existing service consumers.

Solution

Capabilities affected by the decomposition are preserved, while those same capabilities are still allowed to become part of new services. Although the service's original functional context is changed and its official functional boundary is reduced, it continues to provide capabilities that no longer belong within its context or boundary. These are proxy capabilities that are preserved (often for a limited period of time) to reduce the impact of the decomposition on the service inventory (Figure 16.18).

This does not prevent the capabilities in the new services from being independently accessed. In fact, access to the capability logic via its new service contract is encouraged so as to minimize the eventual effort for proxy capabilities to be phased out.

Application

Proxy Capability relies on the application of Service Façade (333) in that a façade is established to preserve affected service capabilities. The only difference is that instead of calling capability logic that is still part of the same service, the façade calls capabilities that are now part of new services (Figure 16.19).

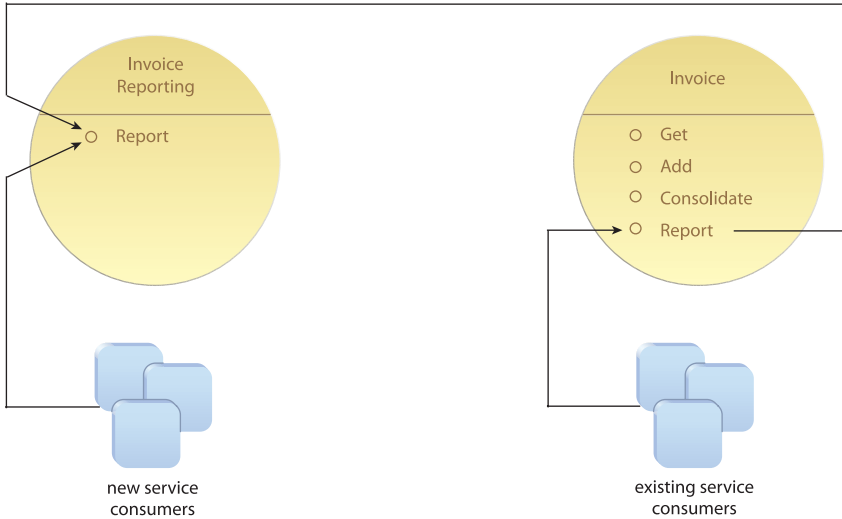


Figure 16.18

By preserving the existing capability and allowing it to act as a proxy for the relocated capability logic, existing consumers will be less impacted.

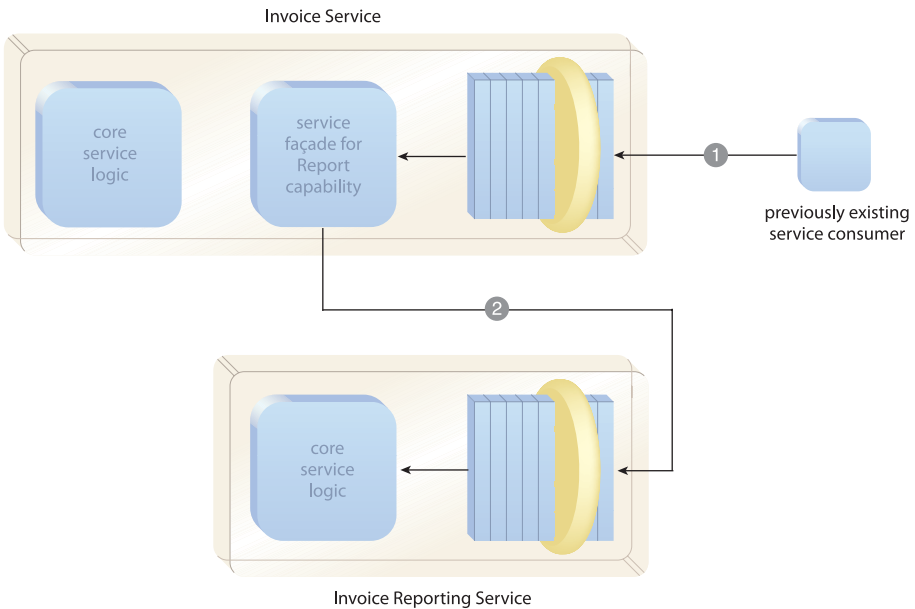


Figure 16.19

When an existing consumer requests an Invoice service operation that has been moved due to the decomposition of the service (1), a newly added façade component relays the request to the capability's new location (2), in this case the Invoice Reporting service.

NOTE

Termination Notification (478) is also commonly applied together with Proxy Capability in order to communicate the scheduled expiry of proxy capabilities.

Impacts

Although the application of this pattern extends the longevity of service contracts while allowing for the creative decomposition of service logic, it does introduce a measure of service denormalization that runs contrary to the goals of Service Normalization (131).

Proxy capabilities need to be clearly tagged with metadata communicating the fact that they no longer represent the official endpoint for their respective logic to avoid having consumers inadvertently bind to them.

Furthermore, this pattern alone does not guarantee that a proxy capability will continue to provide the same behavior and reliability of the original capability it replaced.

Relationships

Whereas Distributed Capability (510) prepares a service for the eventual application of Service Decomposition (489), Proxy Capability actually implements the decomposition while preserving the original service contract.

This is supported by Decoupled Contract (401), which allows the contracts of both the original and the decomposed services to be individually customized in support of the proxy capability. Service Façade (333) also plays an integral role in that it can be used to relay requests (act as the proxy) to and from the newly decomposed service.

And as previously mentioned, this pattern does end up going against the goals of Service Normalization (131). From an endpoint perspective especially, this pattern introduces the appearance of redundant functionality, a trade-off that is accepted in support of service evolution.

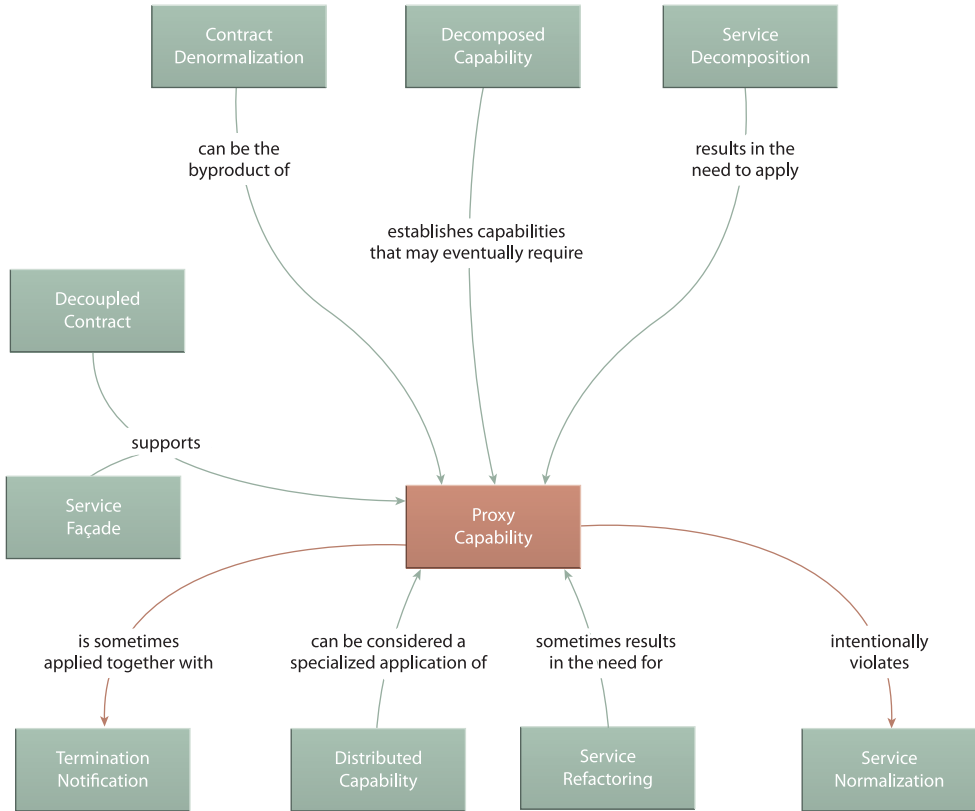


Figure 16.20

Proxy Capability alters the structure of a service in support of the creation of a new service and therefore touches several patterns related to service logic structure and the service decomposition process.

CASE STUDY EXAMPLE

In the case study example for Service Decomposition (489) we explained how the Alleywood team came to the decision to split their existing Employee service into separate Employee and Employee Record services.

To see this through, they need to find a way to achieve the following:

1. Establish a new Employee Record service.
2. Move the corresponding functionality from the Employee service to the new Employee Record service.

- 3. Complete Steps 1 and 2 without changing the original Employee service contract so as to not impact existing service consumers.

To complete Step 1 they model the new Employee Record service, as shown in Figure 16.21.

To accomplish Steps 2 and 3 they employ Proxy Capability for each of the capabilities in the Employee service that needs to be moved to the Employee Record service. Figure 16.22 illustrates how two of the original Employee service capabilities map to four of the Employee Record service capabilities.



Figure 16.21

After some analysis, the new Employee Record service candidate is modeled with four capability candidates.

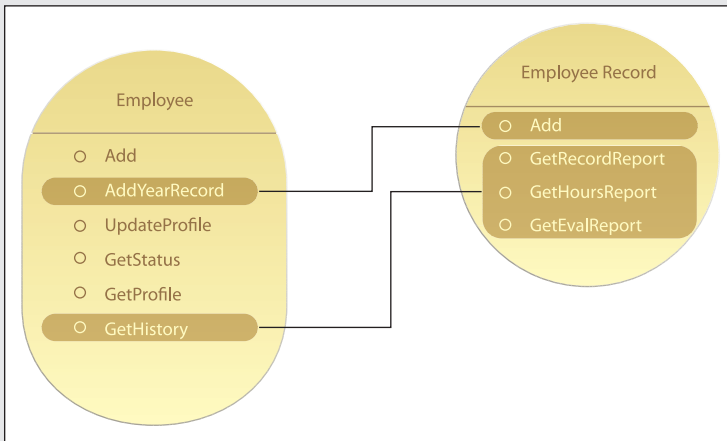


Figure 16.22

The Employee service's AddRecord and GetHistory capabilities are positioned as proxies for the Employee Record's Add, GetRecordReport, GetHoursReport, and GetEvalReport capabilities.

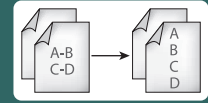
The Employee Record service is eventually designed and delivered as a fully functional, standalone service. However, the Employee service contract remains unchanged, plus additional logic is added in the form of a façade component. This functionality responds to requests for the original AddRecord and GetHistory capabilities and then relays those requests over to the Employee Record. The eventual responses are then received and passed back to the Employee service consumer.

However, one issue remains. In order for the GetHistory operation to work, it must make three calls to the Employee Record service (one to each of the three GetReport operations).

The team considers whether to add a corresponding GetHistory operation to the Employee Record service just for the proxy work that the Employee service must perform. But, they are concerned that the additional operation will be confusing to other consumers. They decide instead to try to accelerate the retirement of the Employee GetHistory operation.

Decomposed Capability

How can a service be designed to minimize the chances of capability logic deconstruction?



Problem	The decomposition of a service subsequent to its implementation can require the deconstruction of logic within capabilities, which can be disruptive and make the preservation of a service contract problematic.
Solution	Services prone to future decomposition can be equipped with a series of granular capabilities that more easily facilitate decomposition.
Application	Additional service modeling is carried out to define granular, more easily distributed capabilities.
Impacts	Until the service is eventually decomposed, it may be represented by a bloated contract that stays with it as long as proxy capabilities are supported.
Principles	Standardized Service Contract, Service Abstraction
Architecture	Service

Table 16.7

Profile summary for the Decomposed Capability pattern.

Problem

Some types of services are more prone to being split after they have been developed and deployed. For example, entity services derive their functional context from corresponding business entities that are documented as part of common information architecture specifications. Often, an entity service context will initially be based around a larger, more significant business entity or even a group of related entities.

This can be adequate for immediate purposes but can eventually result in a number of challenges (Figure 16.23), including the following:

- As the service is extended, many additional capabilities are added because they are all associated with its functional context, leading to a bulky functional boundary that is difficult to govern.
- The service, due to increased popularity as a result of added capabilities or high reuse of individual capabilities, becomes a processing bottleneck.

Despite a foreknowledge of these challenges, it may still not be possible to create a larger group of more granular services because of infrastructure constraints that restrict the size of potential service compositions. Sometimes an organization needs to wait until its infrastructure is upgraded or its vendor runtime platform matures to the point that it can support complex compositions with numerous participating services. In the meantime, however, the organization cannot afford to postpone the delivery of its services.

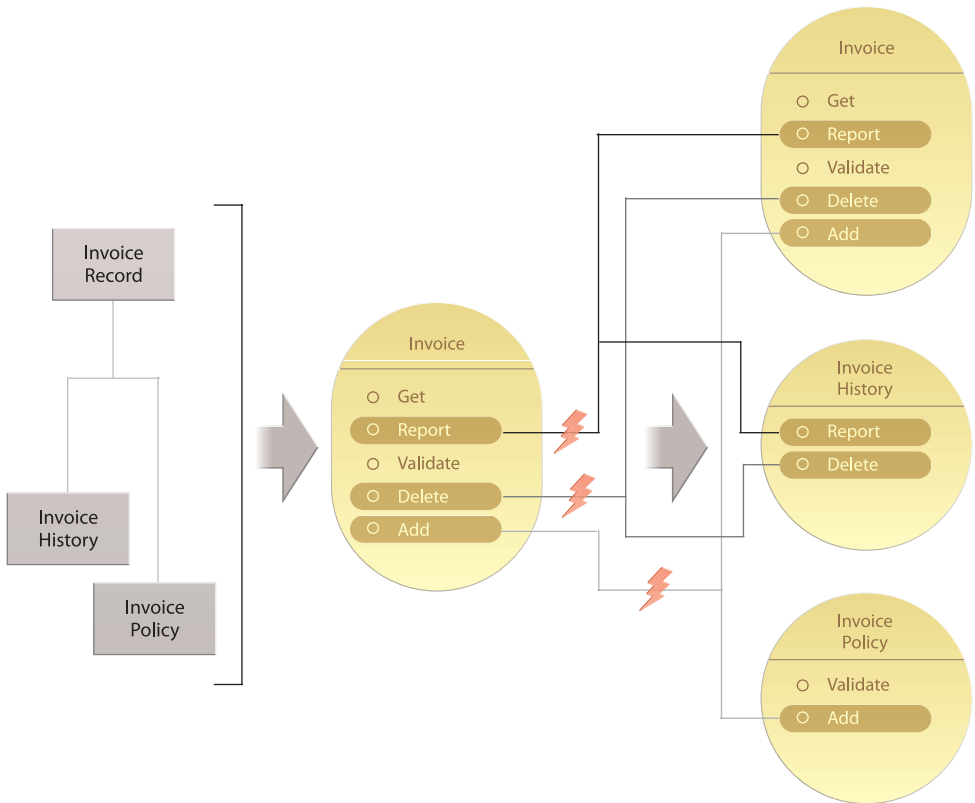


Figure 16.23

An Invoice entity service (middle) derived from a group of Invoice-related business entities (left) exposes coarse-grained capabilities that are difficult to decompose when service decomposition requirements present themselves. Each of the affected Invoice service capabilities needs to be split up in order to accommodate the new services (right).

Solution

Services can be initially designed with future decomposition requirements in mind, which generally translates into the creation of more granular capabilities. With an entity service, for example, granular capabilities can be aligned better with individual business entities. This way, if the service needs to be decomposed in the future into a collection of services that represent individual business entities, the transition is facilitated by reducing the need to deconstruct capabilities (Figure 16.24).

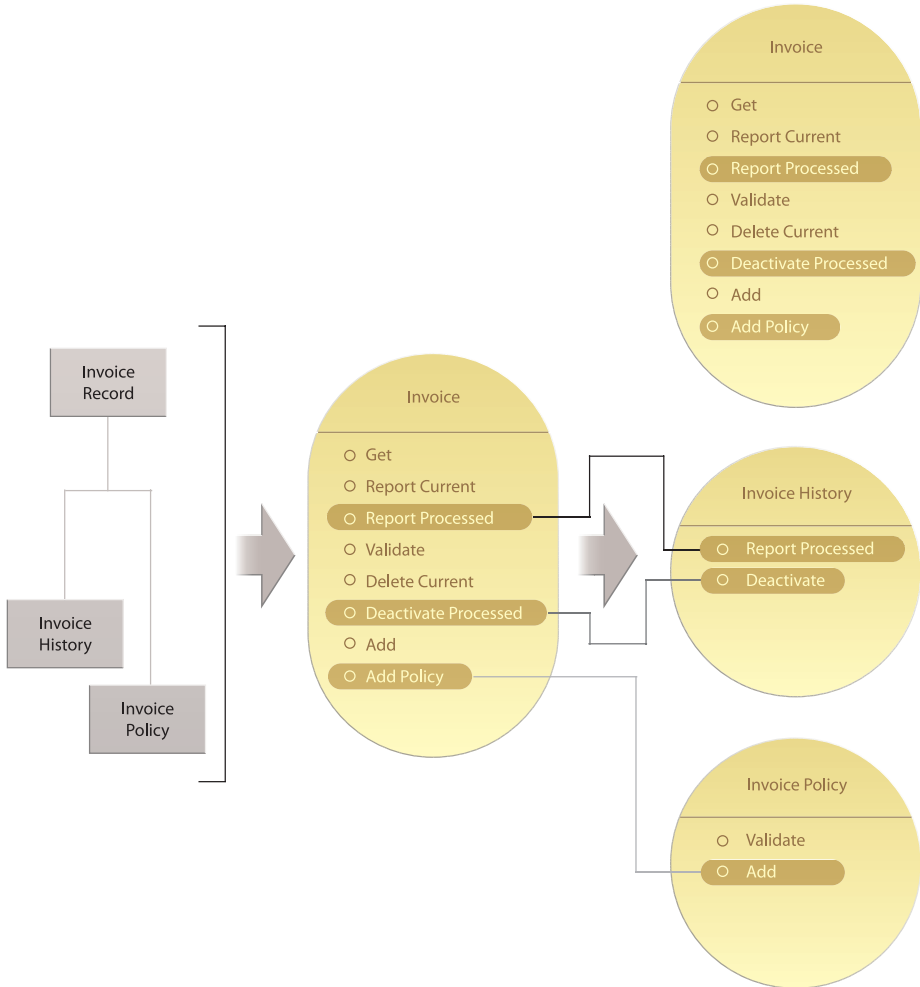


Figure 16.24

The Invoice service (middle) derived from the same business entities (left) introduced in Figure 16.23 now exposes a series of more granular capabilities, several of which correspond directly to specific business entities. This increases the ease at which subsequent service decomposition can be accomplished. The decomposed services (right) are no longer in conflict because the capabilities affected by the decomposition are clearly mapped to the new services. Those same capabilities also remain in the Invoice service contract (top right) as per Proxy Capability (497).

Application

This pattern introduces more up-front service modeling effort in order to determine the appropriate service capability definitions. Specifically, the following considerations need to be taken into account:

- how the current functional scope can potentially be divided into two or more functional contexts
- how capabilities can be defined for these new functional contexts

This modeling effort follows a process whereby a collection of service candidates are defined in association with the scope of the service in question. These service candidates represent future services that can result from a decomposition of the current service and therefore provide a basis for capability candidates to be defined in support of the decomposition.

NOTE

This pattern differs from Contract Denormalization (414) in that the latter introduces redundant, granular capabilities for the purpose of supporting consumer requirements. Decomposed Capability allows for targeted granular capabilities (which may or may not be redundant) in order to facilitate the long-term evolutionary requirements of the service and the service inventory as a whole.

Impacts

The initial service contract that results from applying this pattern can be large and difficult to use. The increased capability granularity can impose performance overhead on service consumers that may be required to invoke the service multiple times to carry out a series of granular functions that could have been grouped together in a coarse-grained capability. This may lead to the need to apply Contract Denormalization (414), which will result in even more capabilities.

Even after the service has been decomposed, the existing consumers of the initial service may still need to be accommodated via proxy capabilities as per Proxy Capability (497), requiring the original service contract to remain for an indefinite period of time.

Also, it is sometimes difficult to predict how a service will be decomposed when initially defining it. There is the constant risk that the service will be populated with fine-grained capabilities that will never end up in other services and may have unnecessarily imposed performance burden upon consumers in the meantime.

Relationships

The key relationship illustrated in Figure 16.25 is between Decomposed Capability and Service Decomposition (489) because this pattern is applied in advance with the foreknowledge that a service will likely need to be decomposed in the future. It can therefore also be viewed as a governance pattern in that its purpose is to minimize the impact of a service's evolution. For this same reason, it relates to Proxy Capability (497) that will usually end up being applied to one or more of the capabilities decomposed by this pattern.

As already mentioned, the more fine-grained capabilities introduced by this pattern may require that Contract Denormalization (414) also be applied.

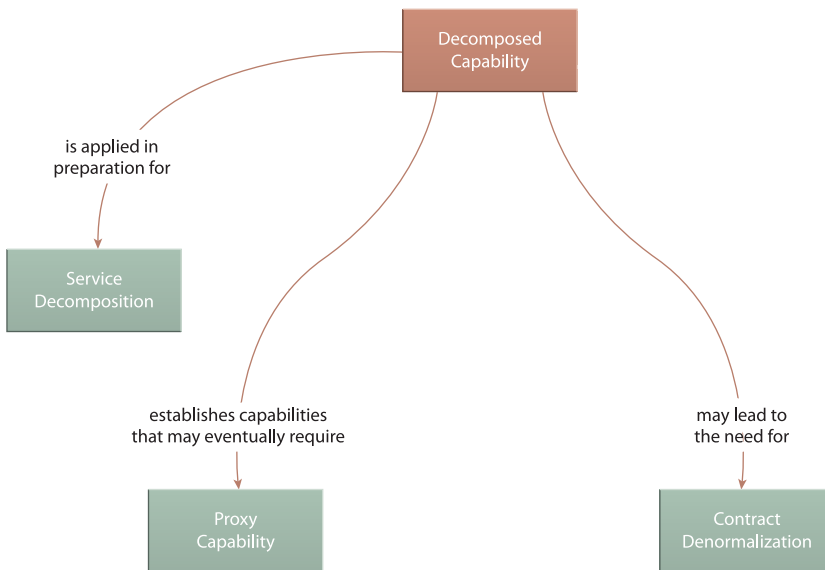


Figure 16.25

Decomposed Capability prepares a service contract for eventual decomposition, making it closely related to patterns associated with Service Decomposition (489).

CASE STUDY EXAMPLE

The case study example for Proxy Capability (497) demonstrated how the decomposition of a service can lead to subsequent design issues, even when establishing capabilities that act as proxies for existing consumers. If the capabilities for the newly derived service don't cleanly match the functional context and granularity of the capabilities of the original service, then awkward and inefficient proxy mapping may result. Depending on how

long the retirement of old capabilities can take, the decomposition of a service can actually increase some of the functional burden it was intended to improve.

Let's focus again on the Employee and Employee Record services explained in the preceding example. If we step back in time when the Employee service was first modeled, we can give the architects and analysts responsible for defining the original service candidate the opportunity to apply Decomposed Capability before proceeding with the physical design and implementation of this service.

In the case of Alleywood, the service would have been based on the two already discussed business entities (Employee and Employee Record) plus a third existing employee-related business entity called Employee Classification. These entities would have determined the capability definition from the beginning in that the original Employee entity service would essentially be viewed as three entity services bundled into one.

Capabilities for this service would have been defined with future decomposition in mind, and the result would have looked a lot like Figure 16.26.

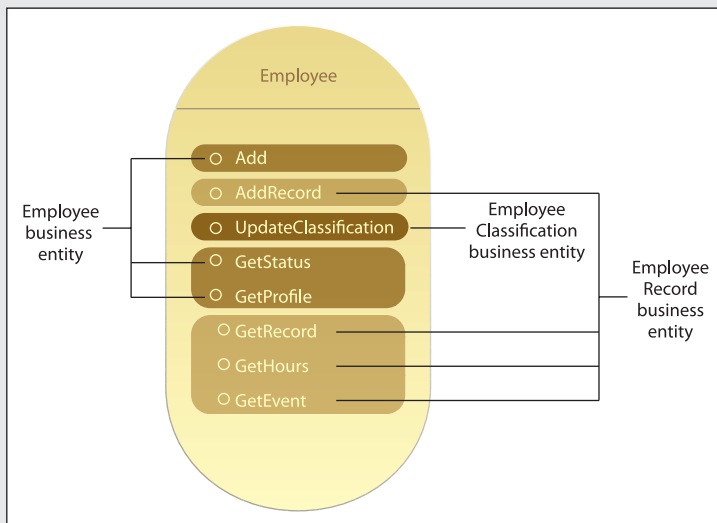
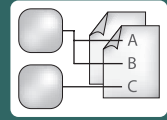


Figure 16.26

The original Employee service modeled to accommodate future decomposition by containing capabilities directly associated with known employee-related business entities. Note that not all of the capability names need to be the same as they will be when the service is decomposed into derived services.

Distributed Capability

How can a service preserve its functional context while also fulfilling special capability processing requirements?



Problem	A capability that belongs within a service may have unique processing requirements that cannot be accommodated by the default service implementation, but separating capability logic from the service will compromise the integrity of the service context.
Solution	The underlying service logic is distributed, thereby allowing the implementation logic for a capability with unique processing requirements to be physically separated, while continuing to be represented by the same service contract.
Application	The logic is moved and intermediary processing is added to act as a liaison between the moved logic and the main service logic.
Impacts	The distribution of a capability's logic leads to performance overhead associated with remote communication and the need for new intermediate processing.
Principles	Standardized Service Contract, Service Autonomy
Architecture	Service

Table 16.8

Profile summary for the Distributed Capability pattern.

Problem

Each capability within a service's functional context represents a body of processing logic. When a service exists in a physically implemented form, its surrounding environment may not be able to fully support all of the processing requirements of all associated capabilities.

For example, there may be a capability with unique performance, security, availability, or reliability requirements that can only be fulfilled through specific architectural extensions and special infrastructure. Other times, it is the increased processing demands on a single capability that can tax the overall service implementation to such an extent that it compromises the performance and reliability of other service capabilities (Figure 16.27).

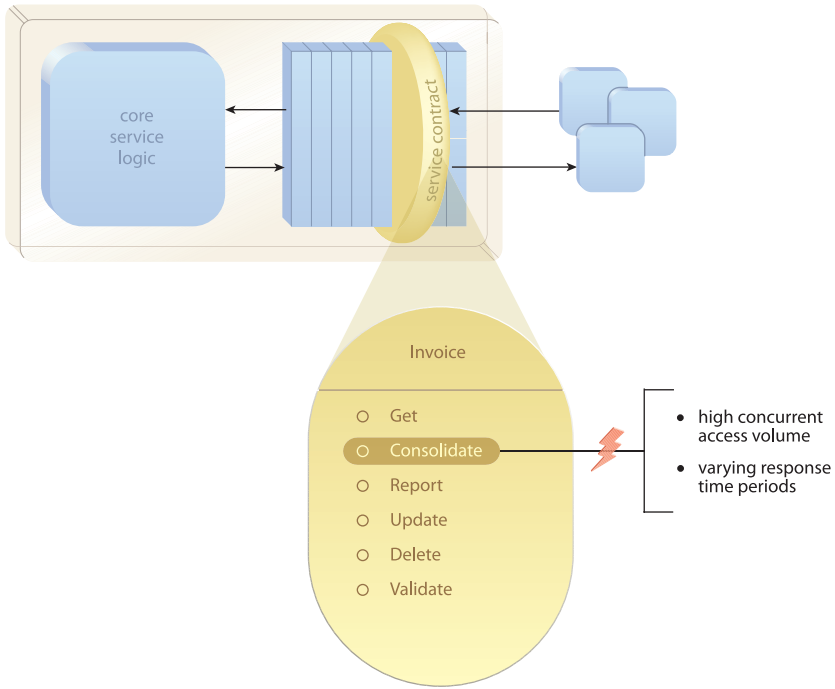


Figure 16.27

The Consolidate operation of the Invoice Web service is subject to high concurrent usage and long response periods when it is required to perform complex consolidation calculations. These factors regularly lock up server resources and therefore compromise the performance and reliability of other service operations.

The logic supporting such a capability can be split off into its own service implementation. However, this would result in the need to break the original functional context for which the service was modeled.

Solution

Capability logic with special processing requirements is distributed to a physically remote environment. Intermediate processing logic is added to interact with local and distributed service logic on behalf of the single service contract (Figure 16.28).

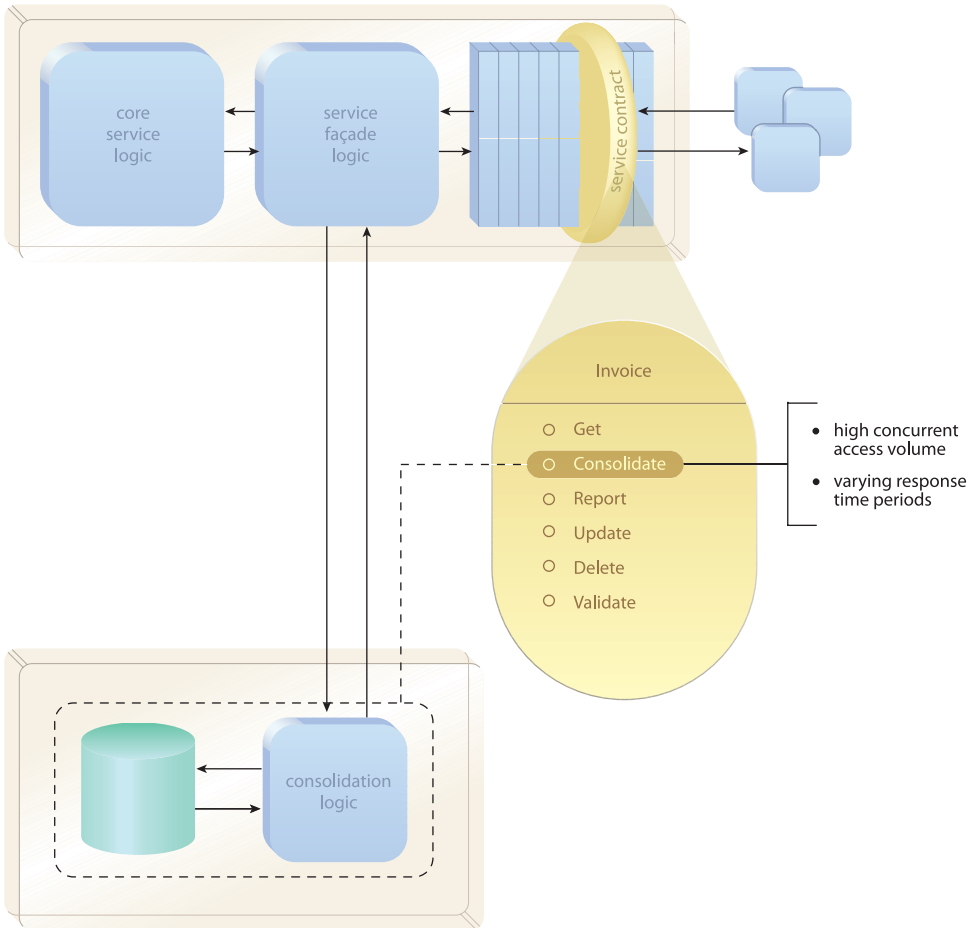


Figure 16.28
The logic for the Consolidate operation is relocated to a separate physical environment. A service façade component interacts with the consolidation logic on behalf of the Invoice service contract.

Application

This pattern is commonly realized through the application of Service Façade (333) in order to establish the intermediate logic that essentially acts as the controller of a “component composition.” The component(s) representing the distributed capability logic interact with the façade logic via remote access.

Performance requirements can be somewhat streamlined by embedding additional processing logic within the façade so that it does more than just relay request and response message values. For example, the façade logic can contain routines that further parse and

extract data from an incoming request message so that only the information absolutely required by the distributed capability logic is transmitted.

An alternative to using Service Façade (333) is Service Agent (543). Event-driven agents can be developed to intercept request messages for a specific service capability. These agents can carry out the validation that exists within the corresponding contract (or perhaps this validation is deferred to the capability logic itself) and then simply route the request message directly to the capability. The same agents can process the outgoing response messages from the capability as well.

Impacts

This pattern preserves the purity of a service's functional context at the cost of imposing performance overhead. The positioning of the contract as the sole access point for two or more distributed implementations of service logic introduces an increased likelihood of remote access whenever the service is invoked.

If the capability logic was separated to guarantee a certain response time during high volume usage, then this may be somewhat undermined by the remote access requirements. On the other hand, overall service autonomy tends to be positively impacted as the autonomy level of the separated capability logic can be improved as a result of its separation.

Relationships

When structuring a service to support distributed capability processing, the service implementation itself exists like a mini-composition, whereby a façade component takes on the role of both component controller and single access point for the distributed service logic. This is why this pattern has such a strong reliance on Service Façade (333) and why it is supported by Decoupled Contract (401) in particular.

Contract Centralization (409) is also an essential part of the service design because it ensures that the contract will remain the sole access point, regardless of the extent the underlying logic may need to be distributed.

When a distributed capability needs to share access to service-related data, Service Data Replication (350) can be employed to help facilitate this access without the need to introduce intra-service data sharing issues. Additionally, this pattern is often the result of applying Service Refactoring (484) and can therefore be considered a continuation of a refactoring effort, especially when applied after the service's initial deployment.

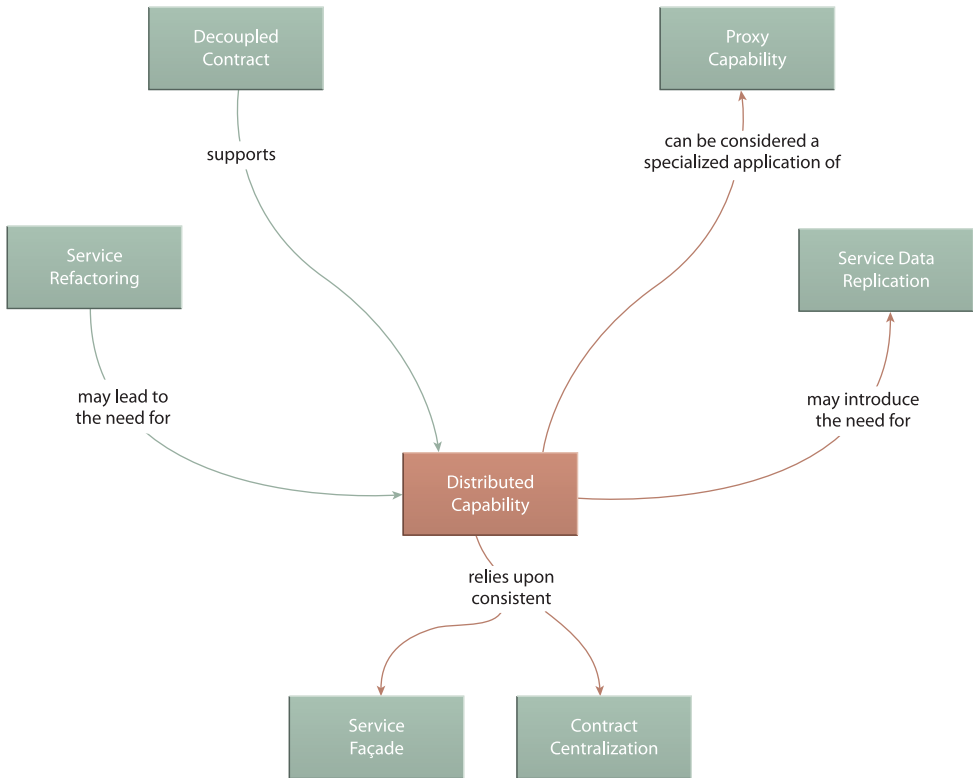


Figure 16.29

Distributed Capability supports the internal decomposition of service logic and therefore has relationships with both service logic and contract-related patterns.

CASE STUDY EXAMPLE

The newly deployed Employee Record service that was defined as a result of applying Service Decomposition (489) and Proxy Capability (497) (see the corresponding case study examples) has become increasingly popular. It is currently being reused within eight service compositions and a new development project is going to be requesting its participation in yet another composition.

For this next composition, the project team is asking that new functionality be added to allow the service to produce highly detailed reports that include various record details and statistics relating to employee hours and ratings from past evaluations. To accommodate this requirement, a new capability is added, called GetMasterReport.

This capability is designed into the Web service contract as an operation that is able to receive parameterized input messages and output large documents comprised of various statistical information and record details.

Preliminary tests show that some of the “from” and “to” value ranges accepted by the operation can take minutes to process because the underlying logic is required to access several databases and then perform a series of calculations before it can produce the required consolidated report.

There are concerns that this one capability will tie up the service too often so that its overall scalability will decrease, thereby affecting its reliability. As a result, the team decides to separate the logic for the GetMasterReport operation to a dedicated server. The Employee Record service is equipped with a façade component that relays requests and responses to and from the separated MSTReportGenerator component.

NOTE

No diagram is provided for this example because the service architecture would be portrayed almost identically to the Invoice service example from Figure 16.28.

This page intentionally left blank

Index

A

agility. *See increased organizational agility, strategic goal*

agnostic, origin of the word, 166

Agnostic Capability design pattern, 140, 295, 316, 725, 744
profile, 324-329

Agnostic Context design pattern, 140, 219, 321, 727, 728
profile, 312-317

agnostic logic, non-agnostic logic, compared, 166-167

Agnostic Sub-Controller design pattern, 606, 620
profile, 607-615

Alexander's pattern language, 90-91

Alexander, Christopher, 89

Alleywood Lumber case study. *See case studies, Alleywood Lumber*

analogies

architecture, 27

design patterns, 3, 87

infrastructure, 27

software programs, 28

application architecture, defined, 28

application section in pattern profiles
described, 103

architecture

application architecture, 28

component architecture, 28

enterprise technology
architecture, 28

hardware versus software
architecture, 30

integration architecture, 28

service architecture. *See service architecture*

service inventory architecture. *See service inventory architecture*

service-oriented architecture.

See SOA

service-oriented enterprise

architecture. *See service-oriented enterprise architecture*

software program. *See software program*

technology architecture. *See technology architecture*

technology infrastructure. *See technology infrastructure*

asymmetric cryptography, 644

asymmetric signatures, 652

Asynchronous Queuing design pattern, 92, 532, 561, 602
profile, 582-591

Atomic Service Transaction design pattern, 92, 587, 606, 635
profile, 623-630

Compensating Service Transaction, 606, 701
profile, 631-638

authentication. *See Brokered Authentication design pattern; Direct Authentication design pattern; Trusted Subsystem design pattern*

authenticity (of message senders), 651

author
about, 783
contact Web site, 13
other books by, 783

B

BEA (Business Enterprise Architecture), 732

behavior correction in service façade logic, 336

blueprints. *See service inventory blueprints*

BMA (Business Mission Area), 732

BOE (Business Operating Environment) compound pattern, 733-734
design principles and patterns in, 734-738
future of, 739

books, related to this book, 7, 90-94

Brokered Authentication design pattern, 391, 640
profile, 661-667

business agility. *See increased organizational agility, strategic goal*

business and technology alignment. *See increased business and technology alignment, strategic goal*

business community, IT community's relationship with, 79-82, 730

business-driven, SOA characteristic, 52-55, 113

Business Enterprise Architecture (BEA), 732

business entities, defined, 176. *See also entity services*

business logic. *See entity services; task services*

Business Mission Area (BMA), 732

Business Operating Environment (BOE) compound pattern, 733-734
design principles and patterns in, 734-738
future of, 739

business rules. *See Rules Centralization design pattern; Validation Abstraction design pattern*

C

callback addresses. *See Service Callback design pattern*

candidate patterns, 5, 12, 46, 492, 748.
See also Web sites, SOAPatterns.org

Canonical Data Model design pattern, Canonical Schema pattern
compared, 158

canonical design patterns, list of, 104

Canonical Expression design pattern, 104, 170, 274, 289, 727
profile, 275-279

- Canonical Protocol design pattern, 104,**
161, 227, 229, 234, 240, 535, 687,
690, 736, 745
profile, 150-157
- Canonical Resources design pattern,**
77, 104, 155, 197, 226, 244, 251, 359,
546, 553, 595, 659, 690
profile, 237-241
- Canonical Schema Bus compound
pattern, 699, 721**
profile, 709-710
- Canonical Schema design pattern, 94,**
104, 233, 400, 535, 671, 674, 709,
726, 736, 745
profile, 158-162
- Canonical Versioning design pattern,**
104, 469, 474, 480, 719
profile, 286-291
- capabilities. See service capabilities**
- Capability Composition design pattern,**
71, 91, 198, 322, 327
profile, 521-525
- capability granularity, defined, 107**
- Capability Recomposition design
pattern, 71, 302, 534, 721, 726, 728**
profile, 526-530
- capitalization in design pattern
notation, 100**
- case studies**
 - Alleywood Lumber, 16
 - Agnostic Sub-Controller design
pattern, 612-615*
 - Atomic Service Transaction
design pattern, 629-630*
 - background, 19-20, 114,
374-375*
- Canonical Protocol design
pattern, 157*
- Canonical Schema design
pattern, 161-162*
- Canonical Versioning design
pattern, 290-291*
- Composition Autonomy design
pattern, 620-622*
conclusion, 744-745
- Cross-Domain Utility Layer
design pattern, 270-271*
- Data Confidentiality design
pattern, 646-648*
- Non-Agnostic Context design
pattern, 323*
- Policy Centralization design
pattern, 213-214*
- Protocol Bridging design
pattern, 692-693*
- Schema Centralization design
pattern, 203-206*
- Service Instance Routing design
pattern, 579-581*
- Service Callback design pattern,
571-573*
- Service Encapsulation design
pattern, 310*
- Service Layers design
pattern, 148*
- Utility Abstraction design
pattern, 173-174*
- Cutit Saws**
 - Agnostic Capability design
pattern, 328*
 - Agnostic Context design
pattern, 317*
background, 17-19, 297-298

- Canonical Expression design pattern*, 279
- Capability Composition design pattern*, 524-525
- Capability Recomposition design pattern*, 530
- Compensating Service Transaction design pattern*, 636-638
- conclusion*, 744
- Data Confidentiality design pattern*, 646-648
- Data Origin Authentication design pattern*, 653-655
- Functional Decomposition design pattern*, 303-304
- Messaging Metadata design pattern*, 542
- Non-Agnostic Context design pattern*, 323
- Policy Centralization design pattern*, 213-214
- Protocol Bridging design pattern*, 692-693
- Schema Centralization design pattern*, 203-206
- Service Instance Routing design pattern*, 579-581
- Service Callback design pattern*, 571-573
- Service Encapsulation design pattern*, 310
- Service Layers design pattern*, 148
- Utility Abstraction design pattern*, 173-174
- Forestry Regulatory Commission (FRC)*
 - Asynchronous Queuing design pattern*, 589-591
 - background*, 21-22
 - Brokered Authentication design pattern*, 666-667
 - Canonical Resources design pattern*, 241
 - Canonical Versioning design pattern*, 290-291
 - Compatible Change design pattern*, 470-471
 - conclusion*, 745-746
 - Concurrent Contract design pattern*, 426-428
 - Contract Centralization design pattern*, 413
 - Contract Denormalization design pattern*, 418-420
 - Data Confidentiality design pattern*, 646-648
 - Data Format Transformation design pattern*, 685-686
 - Data Origin Authentication design pattern*, 653-655
 - Decoupled Contract design pattern*, 407-408
 - Direct Authentication design pattern*, 660
 - Dual Protocols design pattern*, 235-236
 - Event-Driven Messaging design pattern*, 604
 - File Gateway design pattern*, 461-462
 - Intermediate Routing design pattern*, 556

- Legacy Wrapper design pattern*, 446-450
- Multi-Channel Endpoint design pattern*, 456
- Partial State Deferral design pattern*, 361
- Partial Validation design pattern*, 365
- Process Abstraction design pattern*, 187-189
- Reliable Messaging design pattern*, 596-598
- Rules Centralization design pattern*, 222
- Service Data Replication design pattern*, 354-355
- Service Façade design pattern**, 91, 93, 263, 332, 406, 407, 417, 421, 425, 444, 446, 454, 487, 498, 500, 512, 513, 610, 724
 - profile, 333-344
- Service Grid design pattern**, 226, 244, 251, 359, 724, 727, 728
 - profile, 254-259
- Service Layers design pattern**, 92, 164, 719, 728
 - profile, 143-148
- catalogs**. *See pattern catalogs*
- centralization design patterns**, listed, 105
- Certificate Revocation Lists (CRLs)**, 665
- chapters**, described, 7-10
- characteristics**. *See design characteristics*
- chorded circle symbol**, 37
- ciphertext (encrypted data)**, 643
- client**. *See service consumer*
- code examples**
 - Application service, 427-428
 - atomic transactions, 630
 - backwards-compatibility, 471
 - Canonical Versioning design pattern, 291
 - COBOL COPYBOOK message fragments, 447
 - denormalized Officer WSDL definition, 420
 - digitally signed message, 655
 - incompatible changes, 477
 - legacy details in SOAP headers, 450
 - message targeted to specific service instance, 581
 - message with state data in reference parameter, 565
 - Officer WSDL definition, 419
 - plaintext message, 647
 - Policy Centralization design pattern, 214
 - purchase order document, 678, 679
 - purchase order XML Schema definition, 677
 - reference parameter with service instance identifier, 580
 - reliable messaging metadata headers, 598
 - request message with callback address, 572
 - response message with correlation identifier, 573
 - response message with wsa:ReferenceParameters construct, 564
 - revised schema definition for EformApplication service contract, 437

- Schema Centralization design pattern, 206
- schema definition for
 - EformApplication service contract, 435
- SOAP header block, 542
- SOAP message with SAML token, 667
- SOAP message, first in series, 564
- termination information, 483
- undo operation, 638
- version number annotations, 475
- WS-BPEL routine for transactions, 637
- WS-Security metadata, 660
- WSDL definition for Main-AST legacy wrapper service, 449
- XSLT stylesheet, 680
- coexistent application (of compound patterns), 699-700**
- color in figures, 11, 97**
- color tabs, 110**
- commercial off-the-shelf (COTS) software, 738**
- communication protocols. See *Canonical Protocol design pattern; Dual Protocols design pattern***
- Compatible Change design pattern, 288, 290, 464, 469, 475, 480, 481**
 - profile, 465-471
- Compensating Service Transaction design pattern 606, 701, 724, 744**
 - profile, 631-638
- component architecture, defined, 28. See also *architecture***
- components, as services, 45**
- composite patterns, compound patterns versus, 698**
- composition architecture. See *service composition architecture***
- Composition Autonomy design pattern, 263, 348, 353, 360, 454, 605, 606, 628**
 - profile, 616-622
- composition-centric, SOA characteristic, 53, 59-60, 113**
- composition controller, 42, 68, 322**
- composition controller capability, 42, 765**
- composition initiator, 42, 322**
- composition member, 42, 68**
- composition member capability, 42, 764**
- composition sub-controller, 42**
- compositions. See *service compositions***
- compound pattern hierarchy figures, explained, 99-100**
- compound patterns**
 - Business Operating Environment, 733-734
 - Canonical Schema Bus. See *Canonical Schema Bus compound pattern*
 - coexistent application of, 699-700
 - composite patterns versus, 698
 - Enterprise Service Bus. See *Enterprise Service Bus compound pattern*
 - Federated Endpoint Layer. See *Federated Endpoint Layer compound pattern*
 - granularity and, 700
 - joint application of, 699-700
 - Official Endpoint. See *Official Endpoint compound pattern*

- Orchestration. *See* *Orchestration compound pattern*
 - Service Broker. *See* *Service Broker compound pattern*
 - Three-Layer Inventory. *See* *Three-Layer Inventory compound pattern*
 - concerns, defined, 301. *See also* *separation of concerns theory*
 - Concurrent Contracts design pattern, 92, 230, 233, 234, 263, 288, 336, 342, 346, 400, 406, 432, 433, 465, 473, 487
 - profile, 421-428
 - conflict symbol, 11
 - constraint granularity, defined, 108
 - consumers. *See* *service consumers*
 - content-based routing, 552
 - Contract Centralization design pattern, 105, 133, 140, 203, 211, 230, 234, 278, 283, 388, 391, 400, 406, 486, 513, 610, 709, 711, 712, 713, 719, 721, 736, 737
 - profile, 409-413
 - Contract Denormalization design pattern, 134, 342, 400, 406, 425, 470, 471, 507, 508, 634, 721
 - profile, 414-420
 - contract first approach, 752. *See also* *Standardized Service Contract design principle*
 - contract-specific requirements in service façade logic, 336
 - contracts. *See* *service contracts*
 - contributors, about, 784-790. *See also* *Contributors page in front matter*
 - core service logic
 - in components, 45
 - in Web Services, 45
 - service façade logic versus, 335
 - COTS (commercial off-the-shelf) software, 738
 - coupling. *See* *Service Loose Coupling design principle*
 - CRLs (Certificate Revocation Lists), 665
 - Cross-Domain Utility Layer design pattern, 171, 226, 259, 289, 349, 721, 745
 - profile, 267-271
 - CRUD convention, 276
 - cryptography, types of, 644
 - Cutit Saws case study. *See* *case studies, Cutit Saws*
- D**
- Data Confidentiality design pattern, 561, 639, 640, 651, 652, 653, 665, 724, 736
 - profile, 641-648
 - Direct Authentication, 389, 391, 396, 640, 645, 661, 662, 663, 665, 666, 724
 - profile, 656-660
 - Data Format Transformation design pattern, 263, 445, 454, 457, 459, 460, 556, 670, 707, 724
 - profile, 681-686
 - data granularity, defined, 108
 - data integrity (of messages), 651
 - Data Model Transformation design pattern, 94, 99, 127, 159, 160, 263, 445, 454, 457, 459, 460, 671-680, 683, 692, 707, 724, 736
 - Data Origin Authentication design pattern, 393, 561, 640, 644, 645, 659, 724, 736

profile, 649-655

Decomposed Capability design pattern,
170, 464, 492
profile, 504-509

Decoupled Contract design pattern, 91,
93, 278, 342, 400, 411, 412, 417, 425,
486, 500, 513, 709, 719, 721
profile, 401-408

design characteristics, defined, 33

design framework, technology
architecture and infrastructure in,
33-34

design granularity, types of, 107-108

design pattern catalog, defined, 89

design pattern language. *See pattern language*

design patterns
candidates. *See candidate patterns*
cross-referenced with architecture
types, 776-781
cross-referenced with design
principles, 768-774
defined, 86-87
design granularity and, 107-108
design principles compared,
106-107
design principles compared,
106-107
historical influences, 89-95
index of patterns, 791
list of, 791. *See also inside front and back covers*
measures of application, 108
notation, 95-100
profiles, explained, 101-103

Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, et al), 6

design principles

in Business Operating Environment
compound pattern, 734-738
defined, 34, 48-50
design pattern cross-reference for,
768-774
design patterns compared, 106-107
inter-relationships, 50
Service Abstraction, 49, 64, 69, 231,
425, 737
defined, 755

Service Autonomy, 49, 64, 132, 348,
353, 620, 737
defined, 758-759

Service Composability, 41, 49, 59,
527, 738
defined, 764-766

Service Discoverability, 49, 274,
282-283, 736-738
defined, 762-763

Service Loose Coupling, 49, 231,
405, 486, 737
defined, 753-754

Service Reusability, 49, 146, 182,
319, 736, 738
defined, 756-757

Service Statelessness, 49, 64,
250, 737
defined, 760-761

Standardized Service Contract, 49,
159, 229, 231, 400, 405, 736, 738
defined, 751-752

design standards

adherence to, 736
defined, 34
design patterns compared, 149

diagrams. *See figures*

digital signatures. *See signatures*

Direct Authentication design pattern,
389, 391, 396, 640, 645, 661, 662,
663, 665, 666, 724
profile, 656-660

discovery process (services), 282

Distributed Capability design pattern,
336, 342, 353, 405, 464, 745
profile, 510-515

DMZ (demilitarized zone). *See Service
Perimeter Guard design pattern*

DoD (U.S. Department of Defense),
patterns at, 732-739

Domain Inventory design pattern, 42,
74, 106, 113, 115, 120, 121, 147,
155, 159, 160, 270, 674, 675, 718,
721, 745
profile, 123-129

E

EAI patterns, 94

encapsulating legacy environments,
720. *See also File Gateway design
pattern; Legacy Wrapper design
pattern; Multi-Channel Endpoint
design pattern*

encryption. *See Data Confidentiality
design pattern*

endpoint references, 576

enterprise-centric, SOA characteristic,
53, 58-59, 113

Enterprise Integration Patterns (Hohpe
and Woolf), 6, 94

Enterprise Inventory design pattern,
74, 113, 115, 127, 129, 147, 155, 159,
160, 718, 721
profile, 116-122

enterprise resources

defined, 58
enterprise-wide resources
compared, 106

Enterprise Service Bus compound
pattern, 10, 99, 212, 221, 264, 445,
461, 554, 555, 589, 596, 601, 603,
675, 676, 685, 690, 692, 698, 699,
707, 709, 724, 728
profile, 704-706

enterprise technology architecture,
defined, 28

enterprise-wide resources, enterprise
resources compared, 106

Entity Abstraction design pattern, 120,
140, 146, 147, 163, 164, 171, 184,
284, 315, 316, 318, 348, 411, 432,
444, 494, 607, 715, 725, 727
profile, 175-181

entity service layer, 177. *See also Entity
Abstraction design pattern*

entity service model, defined, 164. *See
also Entity Abstraction design pattern*

entity services, service context
definition, 177-178. *See also Entity
Abstraction design pattern*

ESB. *See Enterprise Service Bus
compound pattern*

ESB Architecture for SOA, 7, 601, 700

Event-Driven Messaging design
pattern, 92, 460, 532, 571, 584, 629,
724, 746
profile, 599-604

examples. *See case studies; code examples*

Exception Shielding design pattern,
374, 385, 396, 398, 745
profile, 376-380

F

- façades. *See Service Façade design pattern*
- Federated Endpoint Layer compound pattern, 699, 711, 719
 - profile, 713-714
- federation. *See Federated Endpoint Layer compound pattern; Increased Federation strategic goal*
- figures.
 - color. *See color, in figures*
 - for design pattern notation, 96-100
 - poster. *See poster Web site*
 - symbols. *See symbols*
 - Visio Stencil. *See Visio Stencil*
- File Gateway design pattern, 440, 444, 457-462, 724
- Flexible versioning strategy, 288
- Forestry Regulatory Commission case study. *See case studies, Forestry Regulatory Commission*
- functional context. *See service contexts*
- Functional Decomposition design pattern, 310, 725, 744
 - profile, 300-304

G—H

- glossary Web site, 12, 26, 42
- granularity. *See design granularity*
- grid. *See Service Grid design pattern*
- hardware accelerators, 761
- hardware architecture, 30
- hardware infrastructure, 30
- HMAC (Hashed Message Authentication Code), 651, 658
- How to Solve It (Polya)*, 91
- hybrid architectures, 79

I

- icons in pattern profiles, 101
- impacts section in pattern profiles, 103
- implementation mediums for services, 44
 - components, 45
 - Web services, 45
 - REST services, 46
- information assurance, 736
- information hiding, 737
- infrastructure. *See architecture; technology*
- increased business and technology alignment, strategic goal
 - defined, 51
 - patterns related to, 725-726
- increased federation
 - defined, 51
 - patterns related to, 718-720
- increased intrinsic interoperability
 - defined, 51
 - patterns related to, 721-722
- increased organizational agility
 - defined, 51
 - patterns related to, 728-729
- increased ROI
 - defined, 51
 - patterns related to, 727-728
- increased vendor diversification options
 - defined, 51
 - patterns related to, 723-725
- infrastructure integration architecture, defined, 28
- inter-business service architecture, defined, 78
- Intermediate Routing design pattern, 94, 532, 700, 704, 724, 746
 - profile, 549-556

interoperability. *See increased intrinsic interoperability, strategic goal*

inventory architecture. *See service inventory architecture*

inventory boundary patterns, 112, 114-115. *See Domain Inventory design pattern; Enterprise Inventory design pattern*

Inventory Endpoint design pattern, 78, 92, 121, 127, 226, 342, 348, 396, 455, 620, 675, 691
 profile, 260-266

inventory governance patterns.
See Canonical Expression design pattern; Canonical Versioning design pattern; Metadata Centralization design pattern

inventory standardization patterns, 112, 149. *See also Canonical Protocol design pattern; Canonical Schema design pattern*

inventory structure patterns, 112, 130.
See also Logic Centralization design pattern; Service Layers design pattern; Service Normalization design pattern

IPSec, 391

IT community, business community, relationship with, 79-82, 729-730

J—K

Johnson, Ralph, 107

joint application (of compound patterns), 699-700

KDC (Kerberos Key Distribution Center), 665

Kerberos protocol, 665

Kerberos service accounts, 390

L

layers. *See service layers; Service Layers design pattern*

legacy encapsulation. *See File Gateway design pattern; Legacy Wrapper design pattern; Multi-Channel Endpoint design pattern*

Legacy Wrapper design pattern, 91, 179, 219, 235, 344, 353, 407, 440, 453, 454, 460, 675, 683, 684, 690, 691, 693, 724, 738, 746
 profile, 441-450

logic
 agnostic logic, non-agnostic logic, compared, 166-167
 business logic, utility logic, compared, 166
 service layers and, 167

Logic Centralization design pattern, 97, 105, 113, 130, 133, 147, 178, 183, 234, 280, 283, 284, 354, 412, 523, 711, 712, 713, 719, 727, 737
 profile, 136-140

logic types in service façade components, 336

logical inventory layer patterns. *See Entity Abstraction design pattern; Process Abstraction design pattern; Utility Abstraction design pattern*

loose coupling. *See Service Loose Coupling design principle*

M

MAC (Message Authentication Code), 651

mediator services. *See UI Mediator design pattern*

Message Screening design pattern, 374, 396, 398, 745
 profile, 381-386

messaging, 531-604. *See also* *Messaging Metadata design pattern; Service Messaging design pattern*

Messaging Metadata design pattern, 155, 251, 258, 532, 535, 546, 553, 561, 570, 578, 588, 595, 645, 744
 profile, 538-542

Metadata Centralization design pattern, 92, 105, 138, 140, 142, 274, 278, 289, 724, 727, 737, 745
 profile, 280-285

Minsky, Marvin, 91

mirrored accounts, 390

Multi-Channel Endpoint design pattern, 440, 461, 738, 746
 profile, 451-456

N

naming conventions, standardization of, 275-276

net-centricity, 732

Non-Agnostic Context design pattern, 91, 183-184, 185, 309, 607, 725, 744
 profile, 319-323

non-agnostic logic, agnostic logic, compared, 166-167

non-repudiation, support for, 652

normalization, 90. *See also* *Service Normalization design pattern*

notification service for this book series, 13

O

object-orientation, service-orientation compared, 36

object-oriented design patterns, 91

OCSP (Online Certificate Status Protocol), 665

Official Endpoint compound pattern, 412, 699, 713, 719, 737
 profile, 711-712

Online Certificate Status Protocol (OCSP), 665

open-ended pattern language, defined, 88

operations, defined, 45

orchestrated task services, defined, 186

Orchestration compound pattern, 10, 100, 185, 186, 187, 196, 197, 198, 219, 221, 245, 453, 629, 634, 636, 675, 676, 699, 724, 728
 profile, 701-703

orchestration platform, requirements for, 196

organizational agility. *See increased organizational agility, strategic goal*

P

Partial State Deferral design pattern, 92, 93, 198, 244, 258, 332, 724, 773
 profile, 356-361

Partial Validation design pattern
 profile, 362-365

pattern application sequence figures, 96

pattern application sequences, 88-89

pattern catalogs. *See design pattern catalogs*

pattern sequences, 88-89

pattern language, 88-89. *See also Alexander's pattern language*

pattern profile format, 10, 100-103, 110

pattern relationship figures, 96-98

Pattern-Oriented Software Architecture, Volumes 1–5 (Buschmann, et al), 6

Patterns of Enterprise Application Architecture (Fowler), 6, 93

patterns. *See design patterns*

physical inventory centralization design

patterns. *See Policy Centralization design pattern; Process Centralization design pattern; Rules Centralization design pattern; Schema Centralization design pattern*

PKI (Public Key Infrastructure), 652

plaintext (unencrypted data), 643

policies. *See also Enterprise Service Bus compound pattern; Policy*

Centralization design pattern
collecting requirements for, 210
governance processes needed,
210-211

Policy Centralization design pattern,
105, 133, 192, 219, 220, 412, 432,
724, 744

profile, 207-214

policy enforcement points, 209

policy expressions. *See policies*

Polya, George, 91

poster Web site, 12

Prentice Hall Service-Oriented
Computing Series from Thomas Erl,
11, 13

principle profiles

Service Abstraction, 755
Service Autonomy, 759
Service Composability, 766
Service Discoverability, 763
Service Loose Coupling, 754
Service Reusability, 756-757

Service Statelessness, 761

Standardized Service Contract,
751-752

principles. *See design principles*

problem description in pattern

profiles, 102

Process Abstraction design pattern, 92,
120, 146, 147, 164, 174, 196, 197,
321, 322, 607, 610, 628, 634, 701,
715, 725, 727

profile, 182-189

Process Centralization design pattern,
94, 105, 185, 192, 322, 359, 610,
701, 724

profile, 193-199

profiles. *See pattern profile format; principle profiles*

Protocol Bridging design pattern, 127,
151, 227, 233, 234, 262, 263, 445,
454, 460, 670, 684, 707, 724, 736
profile, 687-693

protocols, defined, 150

Proxy Capability design pattern, 134,
336, 342, 353, 398, 405, 464, 481,
489, 492, 494, 496, 506, 507, 508,
514, 745

profile, 497-503

public key cryptography, 644

Public Key Infrastructure (PKI), 652

public/private key pair, 652

Q–R

queues. *See Asynchronous Queuing design pattern*

recommended reading, 7

reduced IT burden, strategic goal
defined, 51
patterns related to, 729-730

Redundant Implementation design pattern, 140, 230, 233, 234, 259, 263, 332, 353, 365, 454, 495, 619, 620, 665, 727
 profile, 345-349
 references, endpoint, 576
 relationships in compound patterns, 698
 relationships section in pattern profiles, 103
 relaying logic, 336
Reliable Messaging design pattern, 532, 561, 570, 571, 584, 589, 602, 724, 746
 profile, 592-598
 requirement statements in pattern profiles, 101
 resources, defined, 237. *See also Canonical Resources design pattern*
REST-inspired design patterns, 5, 233, 748
 reusability. *See increased ROI, strategic goal; Logic Centralization design pattern; Service Reusability design principle*
ROI (return on investment). *See increased ROI, strategic goal*
 routing, types of, 552. *See also Intermediate Routing design pattern; Service Instance Routing design pattern*
Rules Centralization design pattern, 105, 171, 192, 432, 444, 551, 675
 profile, 216-222

S

SAML, 640
 schema, capitalization of the term, 206

Schema Centralization design pattern, 105, 133, 155, 161, 192, 213, 233, 412, 432, 692, 721, 726, 737, 744
 profile, 200-206
Security Token Services (STS), 665
 security. *See Brokered Authentication design pattern; Data Confidentiality design pattern; Data Origin Authentication design pattern; Direct Authentication design pattern; Exception Shielding design pattern; Message Screening design pattern; Service Perimeter Guard design pattern; Trusted Subsystem design pattern*
 separation of concerns theory, 301
 sequences. *See pattern application sequences*
Service Abstraction design principle, 49, 64, 69, 231, 425, 737
 defined, 755
 design pattern cross-reference for, 771
 implementation requirements, 755
 profile, 755
 service activity, 41, 42, 242, 248, 250, 252, 534, 538, 550, 552, 592, 606, 623, 624, 662
Service Agent design pattern, 45, 67, 94, 171, 364, 369, 370, 371, 379, 384, 432, 460, 513, 532, 535, 552, 553, 571, 578, 588, 595, 645, 724
 profile, 543-546
 service agents, defined, 67
 service architecture
 defined, 61-67
 design pattern cross-reference for, 776-778

- Service Autonomy design principle, 49,**
 64, 132, 348, 353, 620, 737
 defined, 758
 design pattern cross-reference
 for, 772
 profile, 758-759
- Service Broker compound pattern, 92,**
 263, 336, 454, 460, 675, 676, 683,
 685, 690, 692, 698, 699, 704, 706,
 709, 736, 738
 profile, 707-708
- Service Callback design pattern, 460,**
 461, 532, 583, 744
 profile, 566-573
- service candidate, defined, 44**
- service capabilities, 38, 68**
- service catalog, service inventory**
 compared, 43
- service client. See service consumer**
- Service Composability design principle,**
 41, 49, 59, 527, 738
 defined, 764
 design pattern cross-reference for,
 773-774
 principle profile, 764-766
- service composition architecture**
 defined, 61, 68-73
 design pattern cross-reference for,
 778-779
- service composition design,**
 terminology, 42
- service compositions, defined, 40-41**
- service consumer, defined, 38-40**
- service contexts. See also Service Layers**
 design pattern; service models
 for entity services. *See Agnostic*
 Context design pattern; Entity
 Abstraction design pattern
 for task services. *See Non-Agnostic*
 Context design pattern; Process
 Abstraction design pattern
 for utility services. *See Agnostic*
 Context design pattern; Utility
 Abstraction design pattern
- service contracts, 65-66, 764. See also**
 Service Abstraction design principle;
 Service Loose Coupling design
 principle; Standardized Service
 Contract design principle
 naming convention standardization,
 275-276
 patterns related to, 400-437
- Service Data Replication design**
 pattern, 332, 348, 365, 371, 444, 513,
 619, 620, 724, 727, 746
 profile, 350-355
- service definition patterns, 296, 311.**
 See also Agnostic Capability design
 pattern; Agnostic Context design
 pattern; Non-Agnostic Context
 design pattern
- Service Discoverability design**
 principle, 49, 274, 282-283, 736-738
 defined, 762
 design pattern cross-reference
 for, 773
 implementation requirements, 763
 principle profile, 762-763
- service discovery process, 282**
- Service Encapsulation design pattern,**
 58, 93, 184, 303, 322, 725, 744
 profile, 305-310
- service façade components, logic types**
 in, 336

- Service Façade design pattern, 91, 93, 263, 331, 332, 333, 364, 406, 407, 417, 421, 424, 425, 444, 446, 454, 487, 498, 500, 512, 513, 610, 683, 724
 - profile, 333-344
- service façade logic, core service logic versus, 335
- service governance patterns. *See governance patterns*
- service granularity, defined, 107
- Service Grid design pattern, 226, 244, 251, 359, 724, 727, 728
 - profile, 254-259
- service identification patterns, 296, 299. *See also Functional Decomposition design pattern; Service Encapsulation design pattern*
- Service Instance Routing design pattern, 532, 561, 744
 - profile, 574-581
- service inventory
 - defined, 42
 - origin of term, 43
 - service catalog compared, 43
- service inventory architecture
 - defined, 61, 74-75
 - design pattern cross-reference for, 779-781
 - origin of term, 43
- service inventory blueprints, 74
- Service Layer design pattern, Service Layers design pattern compared, 143
- service layers. *See also Cross-Domain Utility Layer design pattern; Service Layers design pattern*
 - combinations of, 164-165
 - entity service layer, 177
 - logic types and, 167
 - task service layer, 183-184
 - utility service layer, 169.
- Service Layers design pattern, 92, 113, 164, 607, 610, 715, 719, 725, 728
 - profile, 143-148
- Service Loose Coupling design principle, 49, 231, 405, 486, 737
 - defined, 753
 - design pattern cross-reference for, 769-771
 - principle profile, 753-754
- Service Messaging design pattern, 94, 155, 251, 532, 541, 546, 561, 570, 588, 595, 602, 645
 - profile, 533-537
- service models. *See also Service Layers design pattern*
 - defined, 144, 164
 - inventory layer patterns,
 - correspondence with, 164
 - list of, 164
- Service Normalization design pattern, 97, 113, 130, 140, 142, 147, 203, 211, 263, 280, 280, 283, 284, 311, 412, 415, 417, 486, 494, 500, 523, 713, 719, 727
 - profile, 131-135
- Service Perimeter Guard design pattern, 171, 374, 379, 665, 666
 - profile, 394-398
- service portfolio. *See service inventory*
- service provider, defined, 39-40
- Service Refactoring design pattern, 133, 342, 406, 411, 494, 495, 513, 724, 728, 737
 - profile, 484-488

service registries, 281. *See also*

Metadata Centralization design pattern

service requester. *See service consumer*

Service Reusability design principle, 49,

146, 182, 319, 736, 738, 764

defined, 756

design pattern cross-reference for, 771-772

principle profile, 756-757

Service Statelessness design principle,

49, 64, 250, 737

defined, 760

design pattern cross-reference for, 773

principle profile, 760-761

service-orientation

defined, 36

method of, 48-52

object-orientation compared, 36

result of, 79-82

service-orientation design principles,

48-50. *See also design principles*

service-oriented analysis, defined,

43-44

Service-Oriented Architecture: Concepts, Technology, and Design, 542

service-oriented architecture. *See SOA*

service-oriented community

architecture, defined, 78

service-oriented computing

defined, 35

strategic goals of, 51-52, 80, 718-730

service-oriented enterprise architecture

defined, 61, 76-77

design pattern cross-reference for, 781

services

as components, 45

as Web services, 45

as REST services

defined, 37-38

implementation mediums, 44-46

signatures, types of, 651

SOA

characteristics of. *See SOA design characteristics*

defined, 37, 78

types of, 61-62

design pattern cross-reference for, 776-781

inter-business service architecture, 78

service architecture, 61, 62-67

service composition architecture, 61, 68-73

service inventory architecture, 61, 74-75

service-oriented community architecture, 78

service-oriented enterprise architecture, 61, 76-77

SOA design characteristics

business-driven, 52-55, 113

composition-centric, 53, 59-60, 113

enterprise-centric, 53, 58-59, 113

inventory design patterns and, 113

vendor-neutral, 52, 54-57, 113

SOA Governance, 7, 123, 282, 464

SOA Magazine, The Web site, 12

SOA Principles of Service Design, 4, 7, 17,

32-33, 50-51, 67, 92, 206, 244, 283,

306, 359, 619, 734, 750

SOA with .NET, 7, 45, 46

SOA with Java, 7, 45, 46

- SOA *with REST*, 7, 46, 748
- SOABooks.com, 7, 11, 13
- SOADoD.org, 739
- SOAGlossary.com, 7, 12, 26, 42, 359
- SOAMag.com, 12
- SOAMethodology.com, 43, 76,
120, 321
- SOAPatterns.org, 12, 102, 233,
256, 748
- SOAPosters.com, 12
- SOAPinciples.com, 7, 50, 76, 306,
750, 768
- SOASpecs.com, 12, 560, 626, 634, 640,
652
- software architecture, 30. *See also*
architecture
- software architecture patterns, 93
- software program, defined, 26, 32-33
- solution description in pattern
profiles, 102
- Standardized Service Contract design
principle, 49, 159, 229, 231, 400,
405, 736, 738
defined, 751
design pattern cross-reference for,
768-769
principle profile, 751-752
- standards. *See design standards*
- state data. *See Partial State Deferral*
design pattern; Service Grid design
pattern; Service Statelessness design
principle; State Messaging design
pattern; State Repository design
pattern; Stateful Services design
pattern
- state keys, defined, 256
- state management. *See Partial State*
Deferral design pattern; Service Grid
design pattern; Service Statelessness
design principle; State Messaging
design pattern; State Repository design
pattern; Stateful Services design
pattern
- State Messaging design pattern, 244,
250, 358, 359, 532
profile, 557-565
- State Repository design pattern, 93,
198, 226, 250, 254, 255, 257, 258,
289, 358, 359, 701, 724, 727
profile, 242-247
- Stateful Services design pattern, 93,
171, 226, 244, 254, 255, 258, 259,
358, 359, 715, 727, 746
profile, 248-253
- Steps toward Artificial Intelligence*
(Minsky), 91
- Strict versioning strategy, 288
- structured pattern language
advantages of, 88
defined, 88
- STS (Security Token Services), 665
- style conventions, used in this book,
11, 110
- summary tables in pattern profiles, 102
- symbols. *See also figures*
chorded circle, 37
color in, 11
for components, 45
in design pattern notation, 96
legend, 11
Visio Stencil, 12
- symmetric cryptography, 644

T

- tabs, used in this book, 110
- task service layer, 183-184
- task service model, defined, 164
- technology and business alignment. *See* *increased business and technology alignment, strategic goal*
- technology architecture. *See also architecture*
 - analogy, 27
 - defined, 26-30
 - design framework and, 33-34
 - scope of, 28
- technology coupling, 402
- technology infrastructure
 - analogy, 27
 - defined, 26, 30-32
 - design framework and, 33-34
- Termination Notification design pattern, 288, 464, 467, 468, 469, 500, 746
 - profile, 478-483
- Three-Layer Inventory compound pattern, 147, 164, 699
 - profile, 715-716
- transactions. *See Atomic Service Transaction design pattern; Compensating Service Transaction design pattern*
- transformation patterns. *See Data Format Transformation design pattern; Data Model Transformation design pattern; Protocol Bridging design pattern*
- Trusted Subsystem design pattern, 374, 398, 745
 - profile, 387-393

U

- U.S. Department of Defense, patterns at. *See DoD*
 - UI Mediator design pattern, 332, 374
 - profile, 366-371
 - Understanding SOA with Web Services* (Newcomer, Lomow), 455
 - undo capabilities, 634
 - Utility Abstraction design pattern, 140, 146, 147, 164, 166, 184, 185, 238, 250, 270, 284, 315, 316, 348, 370, 379, 384, 411, 444, 494, 715, 724, 727, 736
 - profile, 168-174
 - utility logic
 - business logic, compared, 166
 - defined, 168
 - utility service layer, 169. *See also Cross-Domain Utility Layer design pattern; Utility Abstraction design pattern*
 - utility service model, defined, 164. *See also Utility Abstraction design pattern*
 - utility services, service context definition, 170
- V**
- Validation Abstraction design pattern, 179, 203, 211, 219, 364, 400, 425
 - profile 429-437
 - validation logic. *See Concurrent Contracts design pattern; Partial Validation design pattern; Service Abstraction design principle; Validation Abstraction design pattern*
 - vendor diversification. *See increased vendor diversification options, strategic goal; vendor-neutral, SOA characteristic*

version control systems, 757
Version Identification design pattern,
 288, 290, 464, 469, 481, 746
 profile, 472-477
vendor-neutral, SOA characteristic, 52,
 54-57, 113
version numbers, Web services and,
 474
versioning, strategies for, 288
Visio Stencil, 12

W

*Web Service Contract Design and
 Versioning for SOA*, 542
Web services
 Canonical Protocol design pattern
 and, 153
 in Decoupled Contract design
 pattern, 403-405
 defined, 45
 version numbers and, 474
**Web Services Choreography
 Description Language
 (WS-CDL)**, 78
Web service-inspired design patterns, 5
Web sites
 SOABooks.com, 7, 11, 13
 SOADoD.org, 739
 SOAGlossary.com, 7, 12, 26,
 42, 359
 SOAMag.com, 12
 SOAMethodology.com, 43, 76,
 120, 321
 SOAPatterns.org, 12, 102, 233,
 256, 748
 SOAPosters.com, 12

SOAPrinciples.com, 7, 50, 76, 306,
 750, 768
 SOASpecs.com, 12, 560, 626, 634,
 640, 652
 WhatIsSOA.com, 7, 51
 www.refactoring.com, 486
 www.thomaserl.com, 13
WS-Addressing, 542, 560, 568
WS-AtomicTransaction, 252, 626
**WS-CDL (Web Services Choreography
 Description Language)**, 78
WS-Context, 560
WS-Coordination, 251, 626
WS-I Basic Profile, 153
**WS-I Sample Application Security
 Architecture document**, 640
WS-Policy, 209, 480, 752
**WS-PolicyAttachments
 specification**, 209
WS-Security, 640, 643, 651-652
WS-Trust, 665
WSDL
 Canonical Expression design
 pattern and, 276
 language, 752

X—Z

X.509 PKI, 391, 664
XML parsers, 761
XML Schema language, 206, 752
XML Schema, spelling, 206
XML-Encryption, 640, 643
XML-Signature, 640, 652
XSLT, 674