

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.

First printing December 2008

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, Sajjas 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
<i>Information Hiding</i>	64
<i>Design Standards</i>	64
<i>Service Contracts</i>	65
<i>Service Agents</i>	67
<i>Service Capabilities</i>	68
Service Composition Architecture	68
<i>Nested Compositions</i>	72
<i>Task Services and Alternative Compositions</i>	73
<i>Compositions and Infrastructure</i>	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
<i>Inter-Business Service Architecture</i>	78
<i>Service-Oriented Community Architecture</i>	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
<i>Pattern Application Sequence Figures</i>	96
<i>Pattern Relationship Figures</i>	96
<i>Compound Pattern Hierarchy Figures</i>	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, Convergence

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

Sajjas Nasir Imran, Infosys

Berthold Maier, Oracle

Hajo Normann, EDS

Wojtek Kozaczynski

Mark Little, Red Hat

Brian Lokhorst, Dutch Tax Office

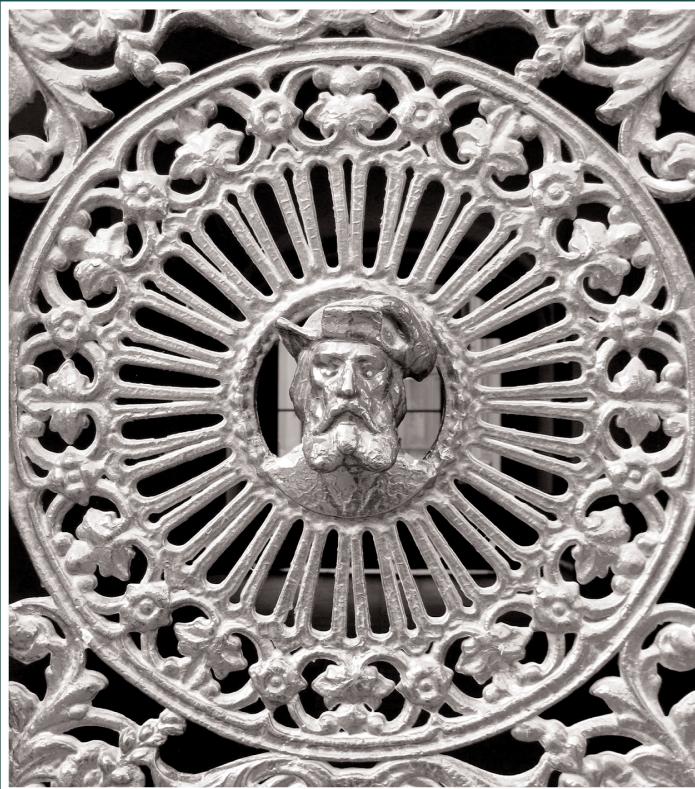
Brian Loesgen, Neudesic

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, SOPERA 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 1



Introduction

- 1.1 Objectives of this Book
- 1.2 Who this Book is For
- 1.3 What this Book Does Not Cover
- 1.4 Recommended Reading
- 1.5 How this Book is Organized
- 1.6 Symbols, Figures, Style Conventions
- 1.7 Additional Information

While recently delivering a week-long workshop at a client location, I was required to spend a fair amount of time waiting in the reception area every day. The client was a very large company in the finance industry, and in order to enter their offices, I had to first request a visitor's pass from the security guard and then wait until someone from the office came down to the lobby to escort me back up.

Upon entering the building for the first time, I noticed that the front door was stuck. It took me two or three tries to force it open. The security guard later told me that a delivery person had accidentally struck the door with some sort of cart, warping the frame and damaging the handle. They weren't expecting replacement parts to be installed for another two weeks and were not allowed to keep the door open.

While waiting for my escort that day, I noticed numerous people (mostly office staff) trying to access the building via the jammed door, each going through the same experience I did. People tried different approaches, some more effective than others. At one point there was an actual line-up impatiently waiting for the person at the front to figure it out. Just about everyone who eventually entered complained about the door to the guard.

On the next day of the workshop I was again waiting in the lobby watching the same story unfold. I saw familiar faces struggling with the door again; getting it ajar seemed more a matter of luck than technique, so it was difficult to remember how one opened it the day before. On this day, the security guard ran toward the door to help people open it whenever he could. However, over time, he found himself rushing back and forth a lot, dealing with the door and tending to people at the reception desk who needed to register and request passes.

The third day came around, and I was surprised to encounter the guard standing outside by the entrance. I could see through the glass walls that someone else was taking care of reception duties. As I approached the door, the guard greeted me, and I assumed he was going to open it to let me in. Instead, he asked me not to enter and proceeded to give me a short lesson on how to open the door with two swift moves. The lesson consisted of a brief explanation and a short demonstration. I thanked him and moved to go inside, but he stopped me, shut the door, and then said "Ok, now you do it." And so I did.

While waiting in the lobby that morning, I watched him give that same lesson to just about everyone who needed to enter. Sometimes the guard had a whole class as a group of office employees who arrived at the same time were taught together.

As I walked toward the building on the fourth day, I noticed the guard was no longer at the door. Recalling “the moves” I’d learned the day before, I proceeded to open it with relative ease. As I entered the lobby, I could see that the same guard was alone again to manage the reception area. Then, while waiting for my escort as usual, I witnessed droves of people entering the building with little to no problems.

It was even more impressive the following day during my last morning. People were coming and going without breaking a stride. It was as if the door had actually been fixed. At the end of that last day of training, I said good-bye to the guard and complimented him on how he dealt with the door issue. “You’re a true problem solver,” I said. “Yes, I know,” he responded with a grin, “I’m the smart one.”

On the taxi ride back to the airport, I thought some more about the damaged door and the solution the guard came up with. I did some rough math, taking into account how long it took the average person to get past the jammed door during the first two days and how many people I saw streaming into the building every morning. I estimated that over the last two days (after each employee was given a lesson by the guard) about 35,000 seconds were saved, translating into around 9.7 hours.

I never did find out when that door was eventually fixed, but assuming it took another week as expected, that time savings could easily be doubled or tripled. That’s potentially 20–30 extra working hours the company received, thanks to one person’s ingenuity.

This experience reminded me of why I felt strongly about putting together this catalog of design patterns. That guard spent the second day trying a variety of ways to deal with a problem until he found a proven method that was effective, easy to learn, and repeatable by anyone. On the third day he transferred that knowledge to all who needed it, and on the remaining days they put that knowledge to good use. In the end, the cumulative benefit was significant because all of the employees who saved time were able to spend that time solving new problems for the benefit of the company.

While problem solving is a fundamental skill we all possess, not everyone should have to solve the same problems. This is the basic rationale behind design patterns. There are jammed doors along the path to completing just about any IT project, perhaps even more so with SOA initiatives simply because their scope tends to be larger and more ambitious. I hope you’ll find this book an effective resource for “learning the moves” to counter problems you might have to face in pursuit of realizing your own service-oriented solutions.

1.1 Objectives of this Book

A design pattern is simply a proven design solution for a common design problem that is formally documented in a consistent manner. This book was written with one primary goal in mind: to provide a master pattern catalog and pattern language for SOA and service-orientation. This sole objective has driven this collection of design patterns through numerous rounds of reviews, revisions, and community participation.

1.2 Who this Book is For

This book is intended for IT practitioners who:

- want to learn proven design solutions and practices for building SOA implementations
- want to prepare themselves for common challenges associated with the definition and design of services and service-oriented solutions
- want to learn about SOA and service-orientation by studying detailed aspects of fundamental design
- want to learn about the different types of service-oriented architectures and understand exactly how they are distinct from other architectural models
- want to gain a deep insight into the complexion of modern-day service-oriented solution design

This book can essentially be considered a reference text for use by anyone involved with the construction of service-oriented solutions.

1.3 What this Book Does Not Cover

The following sections highlight specific subject areas not addressed in this book.

Topics Covered by Other Books

This title is dedicated to documenting design patterns only. Because most of the patterns in this catalog were specifically created in support of service-orientation, there are many cross-references to design principles whenever they are related or relevant to a particular pattern. These design principles are covered separately in *SOA Principles of Service Design*, a companion guide for this book.

Furthermore, this book does not contain a tutorial about Web services or service-oriented computing. There are several publications that have already covered these areas in detail. Suggestions are provided in the upcoming *Recommended Reading* section.

Web Service and REST Service Design Patterns

The Web services technology platform has historically influenced the evolution of service-oriented computing, affecting the complexion and feature-set of typical service-oriented architecture implementations. As a result, numerous design patterns in this book make reference to the use of Web services, and the majority of examples provided show services being implemented as Web services.

Furthermore, a series of REST-inspired design patterns were also developed for this pattern catalog but were not considered ready for inclusion in this first edition of the printed *SOA Design Patterns* book. These patterns have been published in the *Candidate* section of the SOAPatterns.org Web site where they will be subjected to on-going reviews, along with other candidate patterns.

It is important to note that the purpose of this book is to provide a catalog of design patterns that help solve problems specific to the realization of SOA and service-orientation. As has been established in previous series titles, Web services and REST services provide implementation *options* for building services as part of service-oriented solutions.

SOA Standardization Efforts

There are several efforts underway by different standards and research organizations to produce abstract definitions, architectural models, and vocabularies for SOA. These projects are in various stages of maturity, and several overlap in scope.

The mandate of this book series is to provide the IT community with current, real-world insight into the most important aspects of service-oriented computing, SOA, and service-orientation. A great deal of research goes into each and every title to follow through on this commitment. This research includes the detailed review of existing and upcoming technologies and platforms, relevant technology products and technology standards, architectural standards and specifications, as well as interviews conducted with key members of leading organizations in the SOA community.

As of the writing of this book there has been no indication that any of the deliverables produced by the aforementioned independent efforts will be adopted as industry-wide SOA standards. In order to maintain an accurate, real-world perspective, these models and vocabularies can therefore not be covered or referenced in this book.

However, given the unpredictable nature of the IT industry, there is always a possibility that one or more of these deliverables will attain industry standard status at some point in time. Should this occur, this book will be supplemented with online content that describes the relationship of the standards to the content of this book and further maps concepts, terms, and models to whatever conventions are established by the standards. This information would be published on SOABooks.com, as described in the *Updates, Errata, and Resources* page. If you'd like to be automatically notified of these types of updates, see the *Notification Service* section for more information.

NOTE

This comment refers to SOA-related specifications only. There are numerous standards initiatives that have produced and continue to produce highly relevant technology specifications, such as those used for XML and Web services. These are referenced, explained, and otherwise documented wherever appropriate in all series titles.

1.4 Recommended Reading

As already mentioned, this book establishes a master pattern catalog for SOA design patterns. Many of these patterns have roots in the following previously published pattern catalogs that are recommended reading, especially if you are new to the world of design patterns:

- *Design Patterns: Elements of Reusable Object-Oriented Software* (E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley 1994)
- *Patterns of Enterprise Application Architecture* (M. Fowler, Addison-Wesley 2003)
- *Enterprise Integration Patterns* (G. Hohpe, B. Woolf, Addison-Wesley 2003)
- *Pattern-Oriented Software Architecture, Volumes 1–5* (F. Buschmann, K. Henney, M. Kircher, R. Meunier, H. Rohnert, D. Schmidt, P. Sommerlad, M. Stal, Wiley 1996–2007)

How the patterns in these and other publications have influenced the SOA design patterns in this book is further discussed in the *Historical Influences* section in Chapter 5.

While this book includes basic patterns that describe foundational parts of SOA and service-orientation in detail, it does not provide a great deal of introductory coverage of SOA or service-oriented computing as a whole. If you are new to SOA and service-orientation, you can consider reading the following titles that are part of this book series:

- *SOA Principles of Service Design*
- *Service-Oriented Architecture: Concepts, Technology, and Design*

Furthermore, in preparation for those patterns that are focused on design solutions that entail the use of Web service technologies, you can use this book as a companion reference guide:

- *Web Service Contract Design and Versioning for SOA*

Note also that the following additional series titles are in development, each of which further explores and builds upon the SOA design patterns documented in this book:

- *SOA with Java*
- *SOA with .NET*
- *ESB Architecture for SOA*
- *SOA Governance*
- *SOA with REST*

You can check on the availability of these titles at SOABooks.com and you can further read up on fundamental topics pertaining to SOA and service-orientation at WhatisSOA.com and SOAPrinciples.com. Finally, you can take advantage of an online master glossary for this book series at SOAGlossary.com.

1.5 How this Book is Organized

This book begins with Chapters 1 and 2 providing introductory content and case study background information respectively. The remainder of the book is grouped into the following parts:

- Part I: Fundamentals
- Part II: Service Inventory Design Patterns
- Part III: Service Design Patterns
- Part IV: Service Composition Design Patterns
- Part V: Supplemental
- Part VI: Appendices

Part I: Fundamentals

Chapters 3, 4, and 5 in this first part set the stage for all of the design patterns that follow in the subsequent parts by covering key terminology and design issues and by providing an exploration of architecture design principles and the service-oriented architecture types that are later referenced by the individual design pattern descriptions. Also provided is an explanation of how pattern profiles in this book are structured, along with additional sections that cover relevant design topics, such as Web services, service design principle references, and pattern types.

Part II: Service Inventory Design Patterns

“Service inventory” is a term used to represent a collection on independently standardized and governed services. Design patterns associated with the design of the service inventory technology architecture are provided in the following chapters:

- *Chapter 6: Foundational Inventory Patterns* – The baseline design characteristics of a service inventory architecture are addressed by a series of closely related design patterns that are presented in a proposed application sequence.
- *Chapter 7: Logical Inventory Layer Patterns* – How services within a service inventory can be grouped into logical layers is covered by a set of design patterns that represent the most common types of service layers.
- *Chapter 8: Inventory Centralization Patterns* – A set of patterns dedicated to centralizing key parts of a service inventory architecture is provided to build upon the preceding fundamental architectural patterns.
- *Chapter 9: Inventory Implementation Patterns* – This more specialized collection of patterns addresses a variety of implementation design issues and options for service inventory architectures.
- *Chapter 10: Inventory Governance Patterns* – Design patterns relating to the post-implementation governance of a service inventory architecture are provided.

Part III: Service Design Patterns

This part is comprised of a set of chapters specific to the design of services and service architecture:

- *Chapter 11: Foundational Service Patterns* – A set of basic design patterns that help establish fundamental service design characteristics via a suggested application

sequence. Collectively, these patterns form the most basic application of service-orientation within a service boundary.

- *Chapter 12: Service Implementation Patterns* – A collection of specialized design patterns that provide design solutions for a range of service architecture-specific issues.
- *Chapter 13: Service Security Patterns* – These patterns primarily shape the internal logic of services to equip them with security controls that counter common threats.
- *Chapter 14: Service Contract Design Patterns* – A set of design patterns focused on service contract design concerns both from a contract content and architectural perspective.
- *Chapter 15: Legacy Encapsulation Patterns* – How services can encapsulate and interact with legacy systems and resources is addressed by this set of patterns.
- *Chapter 16: Service Governance Patterns* – For services already deployed and in use, these patterns address common governance issues related to typical post-implementation changes.

Part IV: Service Composition Design Patterns

Service composition design and runtime interaction are addressed by the patterns in the following chapters:

- *Chapter 17: Capability Composition Patterns* – A pair of core patterns that establish the basis of service capability composition as it pertains to composition design and architecture.
- *Chapter 18: Service Messaging Patterns* – This large collection of patterns is focused on inter-service message exchange and processing and provides design solutions for a wide range of messaging concerns.
- *Chapter 19: Composition Implementation Patterns* – Service composition architecture design and runtime composition integrity are addressed by these patterns.
- *Chapter 20: Service Interaction Security Patterns* – A set of patterns focused exclusively on security issues pertaining to runtime service interaction and data exchange.
- *Chapter 21: Transformation Patterns* – Design patterns specific to the runtime transformation of messages via intermediary processing layers.

Part V: Supplemental

- *Chapter 22: Common Compound Design Patterns* – Many of the previously documented design patterns can be combined into compound patterns that solve larger, yet still common design problems. This chapter provides examples of some of the more relevant combinations, including Enterprise Service Bus (704) and Orchestration (701).
- *Chapter 23: Strategic Architecture Considerations* – This chapter essentially provides a strategic context for all of the content covered in previous chapters by revisiting the key goals of service-oriented computing and highlighting how the attainment of each individual goal can impact the different SOA types first established in Chapter 4.
- *Chapter 24: Principles and Patterns at the U.S. Department of Defense* – A brief exploration of how service-orientation design principles and key design patterns are used at the DoD in relation to the Business Operating Environment (BOE).

Part VI: Appendices

- *Appendix A: Case Study Conclusion* – The storylines for the three case studies first introduced in Chapter 2 and then further explored in subsequent chapters are concluded.
- *Appendix B: Candidate Patterns* – The pattern review process is highlighted along with an explanation of how patterns still under review are classified as candidates.
- *Appendix C: Principles of Service-Orientation* – Summarized descriptions of the eight service-orientation design principles are provided for reference purposes.
- *Appendix D: Patterns and Principles Cross-Reference* – This appendix organizes design patterns for quick reference purposes as they pertain to service-orientation design principles.
- *Appendix E: Patterns and Architectural Types Cross-Reference* – Design patterns are cross-referenced with the four service-oriented architecture types established in Chapter 4.

Note that an alphabetical listing of all design patterns together with their page numbers is provided on the inside cover of this book.

1.6 Symbols, Figures, Style Conventions

The books in this series conform to a series of conventions, as explained here.

Symbol Legend

This book contains more than 400 diagrams that are labeled as *figures*. The primary symbols used throughout the figures are individually listed in the symbol legend located on the inside of the front cover.

How Color is Used

Most symbols have distinct colors associated with them so that they are easily recognized within the different figures. One exception to this convention is when portions of a figure need to be highlighted for a particular reason. In this case, symbols may be colored in red. The conflict symbol (which looks like a lightning bolt) is always red because it is used to highlight points of conflict.

Data Flow and Directionality Conventions

Some of the figures in this book deviate from traditional conventions associated with depicting data flow. This is further explained in the *Service Consumer* section in Chapter 3.

Pattern Documentation Conventions

Each pattern in this book is documented in a consistent format according to a set of pre-defined notation conventions that are explained in the *Pattern Notation* section in Chapter 5. Note that certain general style conventions are changed subsequent to Chapter 5, as explained on the flipside of the Part II divider page (page 110).

1.7 Additional Information

The following sections describe available supplementary information and resources for the books in the *Prentice Hall Service-Oriented Computing Series from Thomas Erl*.

Updates, Errata, and Resources (www.soabooks.com)

Information about other series titles and various supporting resources can be found at SOABooks.com. You are encouraged to visit this site regularly to check for content changes and corrections.

Visio Stencil (www.soabooks.com)

Prentice Hall has produced a Visio stencil containing the color symbols used by the books in this series. This stencil can be downloaded at SOABooks.com.

Community Patterns Site (www.soapatterns.org)

All of the pattern profile summary tables documented in this book are also published online at SOAPatterns.org, as part of an open site for the SOA community dedicated to SOA design patterns. This site allows you to provide feedback regarding any of the design patterns and you can further submit your own pattern candidates. More information about candidate patterns is provided in Appendix B.

Master Glossary (www.soaglossary.com)

This Web site provides a master online glossary for all series titles. The content on this site continues to grow and expand with new glossary definitions as new series titles are developed and released.

Supplementary Posters (www.soaposters.com)

SOAPosters.com provides a set of color posters available for free download as supplements for the books in this series.

The SOA Magazine (www.soamag.com)

The SOA Magazine is a regular publication provided by SOA Systems Inc. and Prentice Hall and is officially associated with the *Prentice Hall Service-Oriented Computing Series from Thomas Erl*. The SOA Magazine is dedicated to publishing specialized SOA articles, case studies, and papers by industry experts and professionals. The common criterion for contributions is that each explores a distinct aspect of service-oriented computing.

Referenced Specifications (www.soaspecs.com)

Various series titles reference or provide tutorials and examples of industry specifications and standards. The SOASpecs.com Web site provides a central portal to the original specification documents created and maintained by the primary standards organizations.

Notification Service

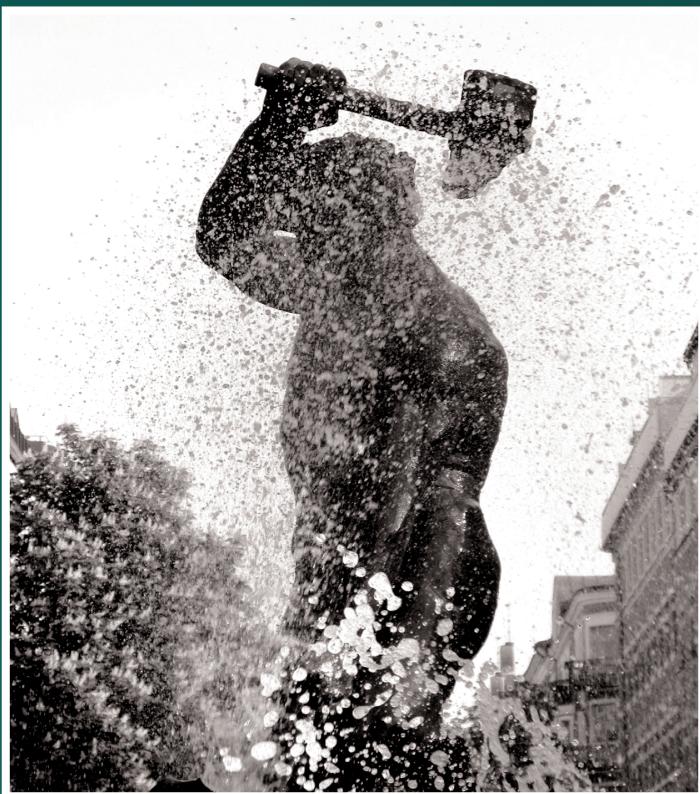
If you'd like to be automatically notified of new book releases in this series, new supplementary content for this title, or key changes to the previously listed Web sites, use the notification form at SOABooks.com.

Contact the Author

To contact me directly, visit my bio site at www.thomaserl.com.

This page intentionally left blank

Chapter 2



Case Study Background

- 2.1 Case #1 Background: Cutit Saws Ltd.
- 2.2 Case #2 Background: Alleywood Lumber Company
- 2.3 Case #3 Background: Forestry Regulatory Commission (FRC)

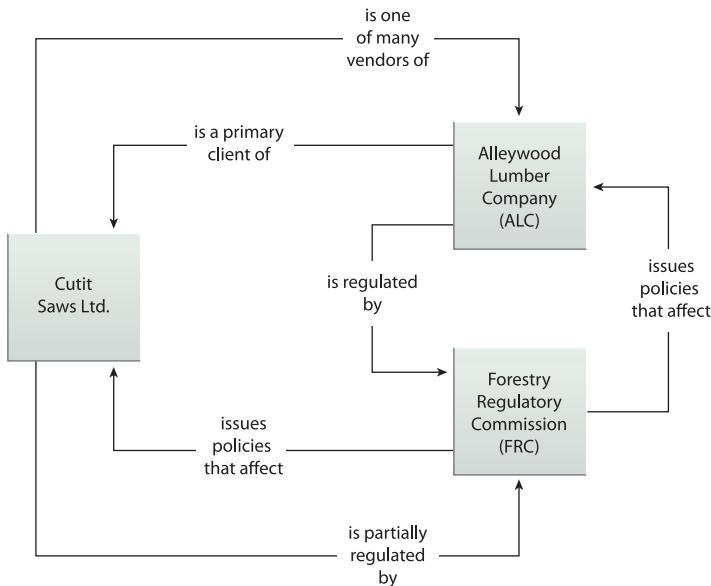
This book covers a range of design patterns that can be incorporated into a variety of environments. Numerous case study examples are provided to supplement design pattern descriptions with some real-world context. All of these examples relate to the background established in this chapter. To make navigation easier, light gray shading has been applied to all case study content subsequent to this chapter.

The upcoming sections provide background information for three case studies based on the following profiles:

- Cutit Saws, a medium-sized private company with a central IT environment, positioning itself as an acquisition target for larger corporations.
- Alleywood Lumber, a large private corporation with a central IT environment and online clients.
- Forestry Regulatory Commission, a large public organization with distributed and partially outsourced IT environments and a very large client base.

As illustrated in Figure 2.1, these organizations have relationships with each other. Cutit Saws is a supplier of chainsaws and chains for Alleywood Lumber, and both of these companies are (to different extents) governed by policies maintained by the Forestry Regulatory Commission.

The next three sections describe each organization's individual set of goals and challenges, establishing the starting point for three separate but intertwined storylines that carry on throughout Parts II, III, and IV of the book and then conclude in Appendix A.

**Figure 2.1**

The three organizations from our case studies have direct and indirect relationships. Cutit and Alleywood have a typical vendor/client arrangement, but the FRC can issue policies that affect both companies.

2.1 Case #1 Background: Cutit Saws Ltd.

Cutit Saws is a niche manufacturer and reseller of high-end hydraulic diamond chainsaws. It has established itself in the tool vendor marketplace by developing a series of unique chainsaw designs that provide effective penetration against especially dense matter.

NOTE

Those of you who read *SOA Principles of Service Design* may already be familiar with Cutit Saws because the Cutit environment formed the basis for a set of detailed examples focused on service design in that book. The background information provided in this section carries forward the Cutit storyline and therefore represents the state of their IT enterprise after the events described in *SOA Principles of Service Design* have transpired.

Note, however, that knowledge of prior examples is not required to understand or work with the case study content in this book.

History

Cutit began as a small company founded by a group of inexperienced business partners based on a set of unique and patented chainsaw blade designs. The effectiveness of these chains eventually led to an unexpected measure of growth, turning Cutit Saws into a successful venture but also placing it into a position it was not prepared for.

Cutit recently released a new diamond blade chain design that became an immediate success. The manufacturing process for this chain is more complex than for previous models, requiring the use of simulations and additional quality assurance steps. As a result, the Cutit team cannot keep up with the demand, and backorders are increasing daily.

With competitors looming and threatening to release similar (reverse-engineered) chain designs, Cutit is under severe time pressure to increase the efficiency and responsiveness of its overall supply-chain process.

Technical Infrastructure and Automation Environment

Much of Cutit's legacy environment is represented by or in some way integrates with a central, custom-developed accounting and inventory management system maintained by a dozen IT staff. This environment was originally designed by one of the company principals but has since become somewhat of an albatross as it has been unable to accommodate the increasing extensibility and scalability demands.

The Cutit team acknowledged the limitations of their modest IT enterprise, and with a realization of how these limitations could severely inhibit the growth potential of the company, they decided to make some changes.

They turned their attention toward SOA and proceeded with the first stage of a larger transition project. The result was the delivery of four services that effectively automated their Lab Project business process, allowing for the efficient simulation of formula applications.

Business Goals and Obstacles

Cutit's primary motivation with their on-going SOA transition project is to establish an IT enterprise that can be more easily extended and modified in response to business change. In recent months, the source of business change has been internal, due to the unanticipated development and success of their newest blade design. Although a fortunate turn of events, this has resulted in chaos for the IT staff.

Another factor that influenced the decision to standardize their environment on SOA is the fact that Cutit owners are planning to position their company as an acquisition target in the coming years. By maximizing revenue generation between now and then, they hope to demand lucrative purchasing terms. They also feel that a standardized and optimized service-enabled IT environment will make the company a more attractive acquisition.

To realize these goals, Cutit continues to focus on their fundamental supply-chain processes. With the Lab Project portion of their manufacturing process completed, they now turn their attention to automating and service-enabling the inventory transfer and back-order fulfillment process that is carried out immediately after chain manufacturing is completed.

2.2 Case #2 Background: Alleywood Lumber Company

Alleywood is a large-sized corporation but still considered a medium-sized contender in the global lumber industry. The company is comprised of three mills distributed in the U.S. and Canada and a head office based out of Chicago. It processes a variety of lumber and supplies for both domestic and international clients.

History

The Alleywood organization has been privately owned by the Alleywood family for three generations. However, recent legislation and foreign trade regulations have made it difficult for it to compete with some of the larger, more internationally established lumber corporations.

The Alleywood family reluctantly agreed to make the company available for sale. Several interested parties emerged, and subsequent to much negotiation and communication, Alleywood was purchased by the McPherson Corporation. Terms were agreed to last year, and the transition is just now beginning to take place.

McPherson is a conglomerate that owns several companies, including the Tri-Fold Paper Mills. One of the reasons McPherson chose Alleywood was to develop its own supply chain from raw wood to refined paper goods.

Although Alleywood retained most of its 900 employees, its upper management was almost completely replaced. A primary goal of the new CIO is to revamp Alleywood's IT environment so that it can be more easily connected to the already services-enabled Tri-Fold enterprise.

Technical Infrastructure and Automation Environment

Alleywood has historically relied on a central ERP system. Outdated by today's standards, it was implemented about eight years ago as part of a very large migration project, replacing many mainframe systems. Since then, the ERP vendor has been assimilated by a larger vendor, and in its current form, the product has become unsupported and obsolete. Due to declining profits over the past two years, no effort was made to upgrade the system.

Business Goals and Obstacles

The focal point of this transition is the collection of central repositories that contain accounting, inventory, financial history, and related corporate data. This information needs to be ported into the new environment in such a manner that the consolidated IT enterprise is still efficient and responsive to business change. For example, Alleywood needs to be constantly able to adapt to regulatory policy changes issued by the FRC. It has been able to do so on its own with relative success but now must continue to adapt to these changes without impacting connectivity with the Tri-Fold environment.

It was decided from the onset that this environment would be developed from the ground up in support of an enterprise-wide SOA. Services will be custom-built and optimized wherever possible. The Tri-Fold environment is already very service-centric, in that services encapsulating ERP modules have been somewhat standardized in alignment with the recently delivered custom services designed to represent other segments of the Tri-Fold enterprise.

Therefore, in addition to building new services for Alleywood, a strategic objective of this transition will be to reuse key services already developed as part of the Tri-Fold service inventory.

Other critical decision points that need to be addressed are:

- Whether the old data models from Alleywood's legacy databases will be preserved or whether the initiative should encompass a remodeling project as part of the overall data export requirements.
- Whether services reused from Tri-Fold should be centrally maintained by Tri-Fold architects or whether they can be evolved independently by Alleywood architects.
- Whether all services will be custom built for Alleywood or whether a new ERP platform will form the basis of the revised enterprise.

2.3 Case #3 Background: Forestry Regulatory Commission (FRC)

There are many independent regulatory commissions that act as objective extensions of the government to administer policies for specific industries. The FRC is dedicated to overseeing commercial activity related to the forestry industry. As part of that role it is responsible for managing and enforcing policies that pertain to private companies involved with the forestry and lumber trade.

History

As an organizational entity, the FRC has been in existence for over 50 years. It has undergone many changes during that time, as its funding has fluctuated and its directives are repeatedly augmented. In its current state, it exists as a relatively autonomous agency with three main administration offices, 112 satellite locations, and a staff of over 20,000.

Each of the main offices represents one of three primary FRC business divisions:

- Policy Management
- Field Support
- Assessments and Appeals

Although they have always worked together to further the overall goals of the FRC, these divisions have been physically isolated and have developed individual corporate cultures. In many ways, these offices exist as independent organizations.

As the forestry industry expands, so must the FRC. It has grown steadily over the past ten years and expects this trend to continue.

Technical Infrastructure and Automation Environment

The FRC has a massive inventory of products, custom-built systems, middleware, and repositories. No one knows for sure how many applications actually exist, but estimates range from 300 to 450. Of these, a large percentage was built by external solution providers. At last count, the FRC has nearly 900 registered IT vendors.

Those automation solution projects that were outsourced began with formal proposal submission processes governed by dedicated review committees. In the past, the primary criteria applied to the assessment of vendor proposals was the claimed expertise, promised delivery timeline, and, of course, the estimated cost.

Other solutions have been developed in-house. With a total IT staff of over 1,000, the FRC has numerous resources at its disposal. Each of the three administration offices has an IT division with its own departmental structure and organizational hierarchy. There are regular meetings between IT managers from all divisions, but outside of that, there is infrequent communication or coordination. Resources are rarely shared between offices, and when they are, strict charge-back policies are in place to ensure that any loan is eventually compensated.

Business Goals and Obstacles

A year ago an annual financial report was delivered to the president of the FRC and its board of directors revealing that operational IT costs have been higher than ever before. This led to a follow-up strategic meeting for which historical reports were prepared, providing an analysis of IT expenses over the past 10 years.

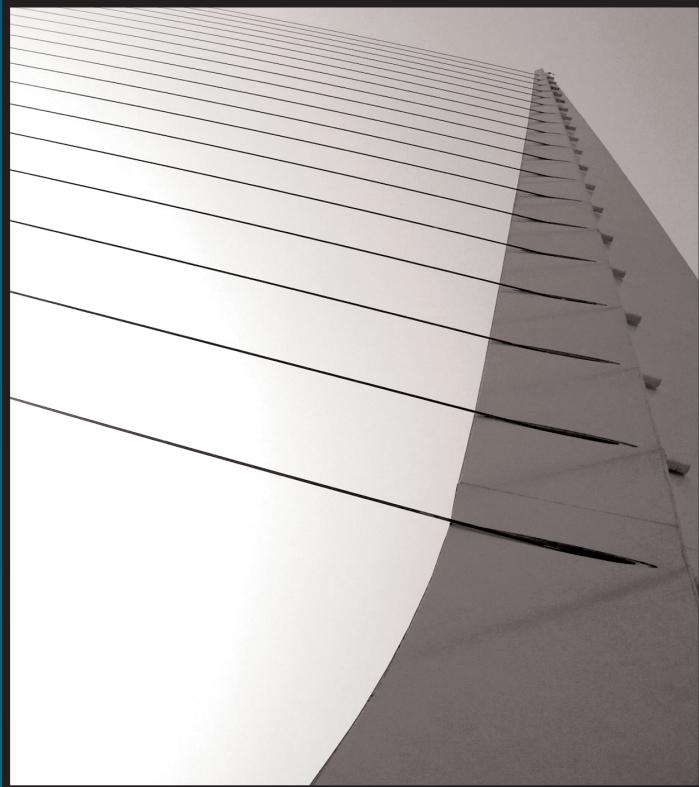
It became evident that, unlike other departments, the cost increases for IT were severely disproportionate with the FRC's rate of growth. For example, five years ago the FRC expanded by 5%. Its IT costs, however, increased by 9%. This last year, the FRC grew by 3%, but the IT budget jumped by 13%. Every year, the three IT divisions are consuming a larger portion of the overall FRC budget. If this alarming trend would be allowed to continue, the FRC would be forced to downsize other departments just to support IT.

Subsequent to a series of tense meetings with the IT directors from each office, an in-depth audit of past IT expenditures was commissioned. The results indicated that every time the FRC expanded or underwent internal policy changes, the underlying IT environments were required to compensate by either building new systems or modifying existing ones. Almost all primary systems are currently integrated with others to some extent. These integration channels were costly to implement and have become especially expensive to change and maintain.

Over time, so many point-to-point integration channels have been created that a change to one can have a significant ripple effect across several others. Therefore, even a minor alteration can result in a costly development, integration, and regression testing effort. This has quietly escalated to a point where IT costs have become unmanageable.

Subsequent to these revelations, the president of the FRC decides that a reorganization is in order, beginning with upper IT management. All IT directors are replaced, and the position of CTO is established. IT divisions will no longer have the independence they've enjoyed so far; now each of the new directors must report to the CTO, and it is up to this individual to figure out how to turn this all around. Following a further independent analysis project, it is decided that a major move toward SOA provides the best option to solving FRC's IT problems.

Part I



Fundamentals

Chapter 3: Basic Terms and Concepts

Chapter 4: The Architecture of Service-Orientation

Chapter 5: Understanding SOA Design Patterns

This page intentionally left blank



Chapter 3

Basic Terms and Concepts

3.1 Architecture Fundamentals

3.2 Service-Oriented Computing Fundamentals

3.3 Service Implementation Mediums

It's been well established that service-oriented computing represents an ambitious platform with the potential to transform the complexion of an enterprise. Design patterns in support of this platform therefore tackle a spectrum of design problems, ranging from the encapsulation of granular functionality to the strategic partitioning of the enterprise into service-enabled domains.

A constant among all the patterns documented in this book is that each, in some shape or form, impacts or relates to technology architecture. A service, a solution comprised of services, a collection of related services—each has a pre-determined architectural design, each claims its own distinct part of the overarching enterprise architecture, and all are collectively designed to work in concert to realize common strategic goals.

Purpose of this Introductory Chapter

This chapter establishes the fundamental links between service-oriented computing, service-orientation, and technology architecture. You will find this content useful if you are new to SOA or if you need to look up definitions to some of the terms used throughout this book. Also, the upcoming architecture-related definitions provide background information for the service-oriented architecture types explained in Chapter 4. Those types are then subsequently referenced in pattern descriptions.

NOTE

This chapter borrows some content from SOAGlossary.com.

3.1 Architecture Fundamentals

To prepare for the upcoming discussion of service-orientation and technology architecture in Chapter 4, let's begin by establishing some fundamental terminology. The next sections establish the following basic architecture-related IT terms:

- *Technology Architecture* – The fundamental physical design of something.
- *Technology Infrastructure* – The underlying, supporting technology environment, including software and hardware.
- *Software Program* – A standalone system that may be a custom-developed application or a purchased product.

We are by no means attempting to formally define these terms for the IT industry. These definitions are simply part of the common vocabulary used by the books in this series to ensure consistency and clarity across all titles. Figure 3.1 further illustrates their meaning.

A Classic Analogy for Architecture and Infrastructure

It's well-known how the IT community borrowed the term "architecture" from its traditional association with the design and construction of buildings and structures. Its origin also helps us establish an analogy that is useful for distinguishing a technology architecture from a technology infrastructure.

A building has a physical design expressed in an architecture blueprint or specification. However, the building exists within a surrounding environment. This environment may or may not provide a lot of support for the building to fulfill its purpose. For example, an office or residential building located within a city is supported by the streets, power plants, utility pole cables, sewer systems, and other resources provided by the city environment. This supporting environment is analogous to technology infrastructure.

In order for a building to take advantage of these infrastructure extensions, its physical design needs to integrate them as part of its official architecture. Therefore, an architecture specification for a building will encompass the parts of its surrounding infrastructure that are relevant to the building. As a result, there is no firm boundary between what constitutes the building architecture and the environmental infrastructure. This same overlap exists in the IT world, as explained in the following definitions.

Technology Architecture

A technology architecture expresses fundamental and foundational aspects of physical design for some piece of technology. Whereas computer hardware products will have their own individual technology architectures, within a typical IT enterprise, we are most concerned with the architecture of software programs.

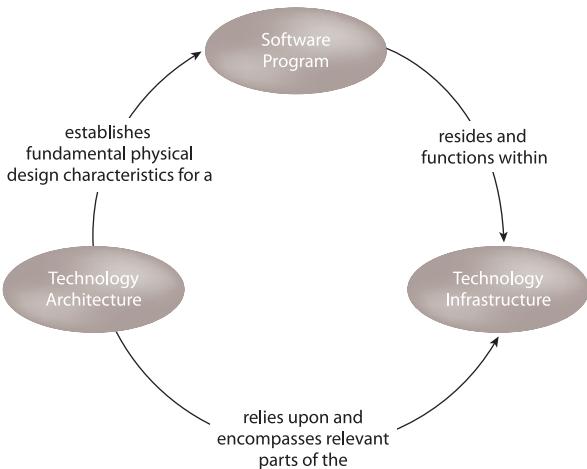


Figure 3.1

An overview of how the enterprise elements represented by these terms relate to each other.

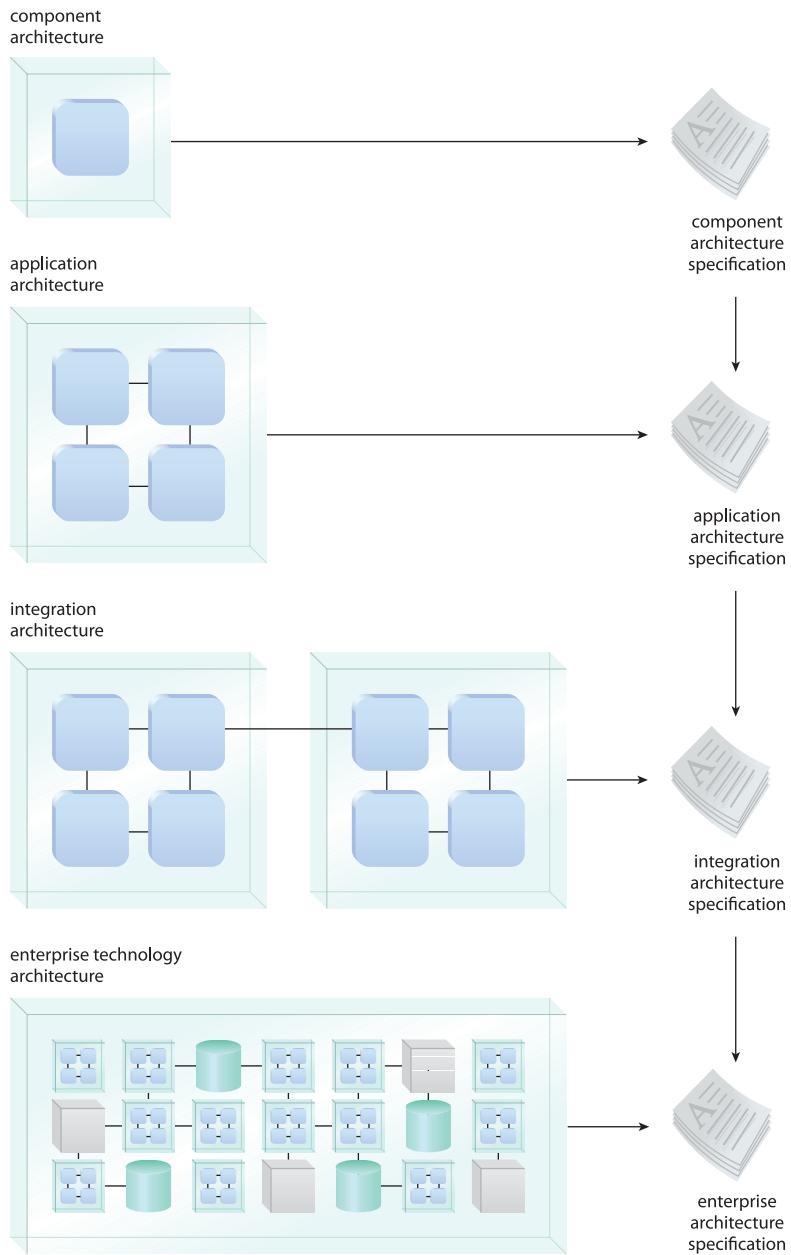
For a program we purchase, we may want to understand its internal design to ensure that it is compatible with the environment already established within our enterprise. For programs we build, it becomes our responsibility to define the physical design ourselves.

When designing a new software program, we need to take into consideration the environment in which it will need to be deployed and in which it will need to carry out its purpose. In most established enterprises, implementation environments already exist in the form of servers, operating systems, and runtime and middleware platforms. As explained in the next section, all of these parts are considered technology infrastructure. And, as with buildings and cities, the software program's architecture is comprised of new parts interwoven with the relevant parts of the infrastructure that already exist as part of the surrounding environment.

The scope of technology architecture can vary depending on what it is we are designing. Some well-known types include:

- *Component Architecture* – In a distributed computing environment, this represents the physical structure of an individual software program that exists as a component.
- *Application Architecture* – A technology architecture with a physical boundary limited to the deployment environment of a particular application or system. In a distributed computing environment, an application architecture can encompass multiple component architectures.
- *Integration Architecture* – The technology architecture of two or more connected applications or systems including whatever technologies, resources, or extensions were added to enable their integration. Many integration architectures include middleware platforms and associated adapter or bridging extensions.
- *Enterprise Technology Architecture* – Unlike component, application, and integration architectures, which are often documented in design specifications *prior* to the creation of programs, enterprise technology architectures frequently result as a documentation of what already exists within an enterprise environment. An enterprise technology architecture specification can encompass (or may just reference) all previously listed forms of architecture and may also act as a formal documentation of the enterprise infrastructure as well.

As shown in Figure 3.2, each of these architecture types represents a different scope, whereby one tends to encompass the other.

**Figure 3.2**

Common traditional levels of documented technology architecture.

In a service-oriented environment the scope of technology architecture can also vary. The distinct forms of service-oriented architectures and how they roughly correspond to the previously listed traditional architecture types is explained separately in the section *The Four Types of Service-Oriented Technology Architecture* in Chapter 4.

NOTE

As just mentioned, you can define an architecture for a piece of hardware or a software program, which is why you will sometimes see the term “architecture” further qualified as hardware architecture or software architecture. Because our focus in SOA projects is primarily on software design, why then do we continue using the broader “technology architecture” term? As revealed by many patterns in this book, architecture in the world of service-oriented computing relies on a combination of software and hardware resources, both of which find their way into typical architecture specifications.

Technology Infrastructure

Within a typical IT enterprise, technology infrastructure represents the environment in which software programs are deployed. As with the term “architecture,” infrastructure can also be qualified with “software” or “hardware” to identify certain parts of this environment.

Common forms of hardware infrastructure include:

- servers and workstations
- routers, firewalls, and networking equipment
- back-up power supplies, cables, and other computer equipment

Types of software typically considered part of an enterprise’s technology infrastructure include:

- operating systems and system APIs
- runtime environments and system-level service agents
- databases and directories
- transaction management programs and message queues
- middleware and adapters
- user account management and security technologies

What generally distinguishes a technology that is part of infrastructure from one that is exclusive to a particular component or application architecture is that it is made available to multiple applications or systems and therefore exists as a resource of the enterprise (and is therefore also separately owned and governed). An example of a common software program that can be either classified as part of infrastructure or specific to an application architecture is a database (Figure 3.3).

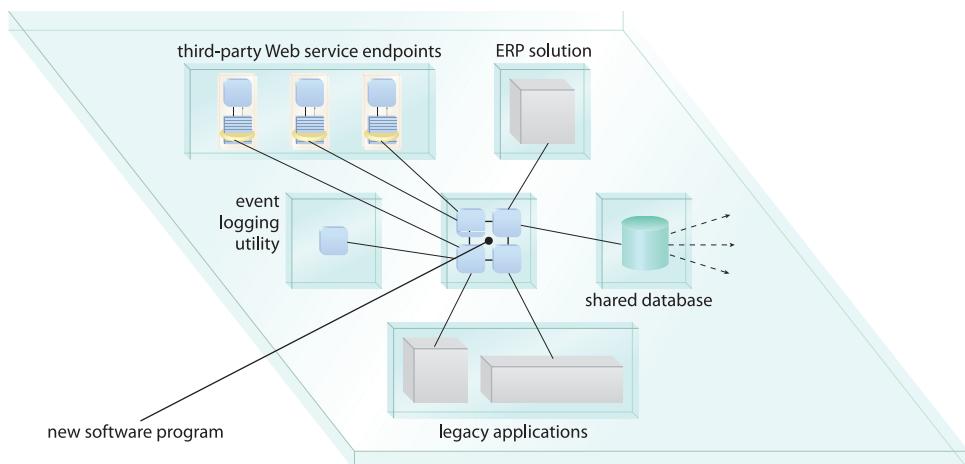


Figure 3.3

A software program implemented within an enterprise finds itself dependent on various resources in the surrounding infrastructure.

As previously mentioned, relevant pieces of technology infrastructure find their way into almost all forms of architecture documentation because they become part of the architecture itself. An enterprise technology architecture specification often documents some or all of an enterprise's infrastructure in a reference format that is made available to authors of other architecture design documents.

The infrastructure of an enterprise will frequently determine the processing potential of technology architectures that reside within it and are built upon it. This potential threshold is then further leveraged or constrained by the design of the architecture itself. Consequently, a software program is required to exist and execute within the boundaries and thresholds established by both its underlying infrastructure and architecture (as explained in Figure 3.4).

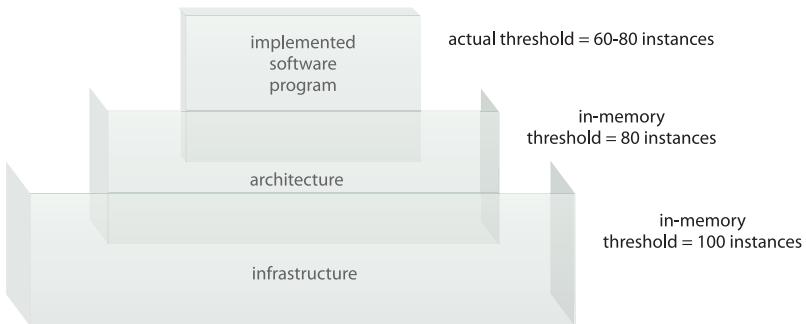


Figure 3.4

Technology infrastructures and architectures collectively establish boundaries that determine the processing thresholds of software programs. In this example, the maximum instances of a program that can be concurrently invoked is less than what the infrastructure can support because of limitations introduced by the architecture and the software program's own implementation.

Software Program

A software program is simply an existing system, application, or solution. It may represent a purchased product or a custom-designed program. In relation to technology architecture, a software program can be considered an implementation of the design documented in an architecture specification, as well as the logic that resides and executes within the supporting environment also specified by the technology architecture.

Part of a software program's design can be documented within an application architecture specification. Usually this part is backend-centric with an emphasis on the program's overall structure (including components it may be comprised of), technologies, and resource requirements. A typical application architecture specification is therefore frequently supplemented with additional types of design documents, such as functional specifications that illustrate the flow and style of the program user interfaces and detailed design documents that establish programming routines and algorithms.

Depending on the conventions, methodologies, or preferences of the IT department, this additional design information may or may not be considered part of the program's official technology architecture (Figure 3.5).

NOTE

The upcoming *Relationship to Design Framework* section briefly provides some reference content for readers of *SOA Principles of Service Design*. If this does not interest you, feel free to skip ahead to the *Service-Oriented Computing Fundamentals* section.

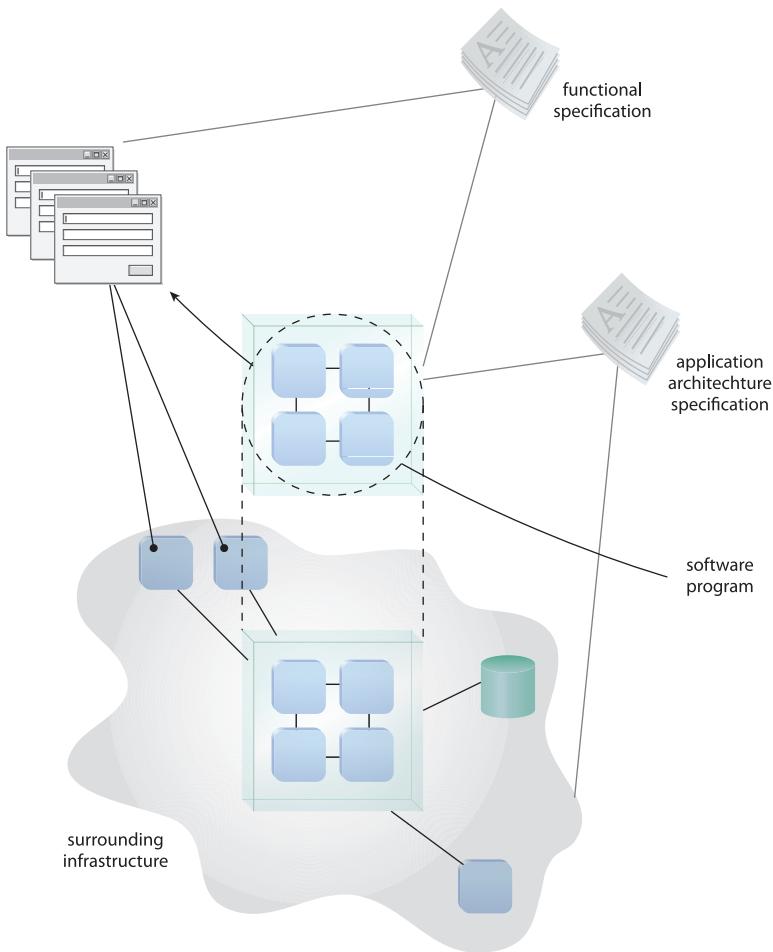


Figure 3.5

The physical design of a software program is partially defined by its application architecture along with relevant parts of the surrounding infrastructure. Other documents, such as a functional specification, establish additional design characteristics, such as the program's user-interfaces.

Relationship to Design Framework

Chapter 3 of *SOA Principles of Service Design* documents a base design framework that includes the following terms:

- *Design Characteristic* – A property of a software program or technology architecture that results from how it was designed. A design characteristic can be any concrete quality, such as the fact that the program is componentized, provides fine or coarse-grained functions, and so on.

- *Design Principle* – An accepted industry practice with a specific design goal. The service-orientation design paradigm is comprised of a set of design principles that are applied together to achieve the goals of service-oriented computing (as explained in the upcoming *Service-Orientation and Technology Architecture* section).
- *Design Pattern* – A proven solution to a common design problem documented in a consistent format. (See the *Fundamental Terminology* section in Chapter 5 for a full definition of this term.)
- *Design Standard* – Design conventions customized individually by organizations in order to reliably deliver solutions in support of the organization’s specific business goals. Design standards can support and optimize the application of design principles and design patterns for particular environments and can help ensure the consistent realization of design characteristics. Conversely, design principles and patterns can form the basis of design standards that are then further customized. (Note that design standards should not be confused with *industry* standards, such as XML and WSDL.)

Figure 3.6 illustrates how closely this design framework can tie into the architecture-related vocabulary we just covered. In the end, principles, patterns, and architecture all revolve around and influence design.

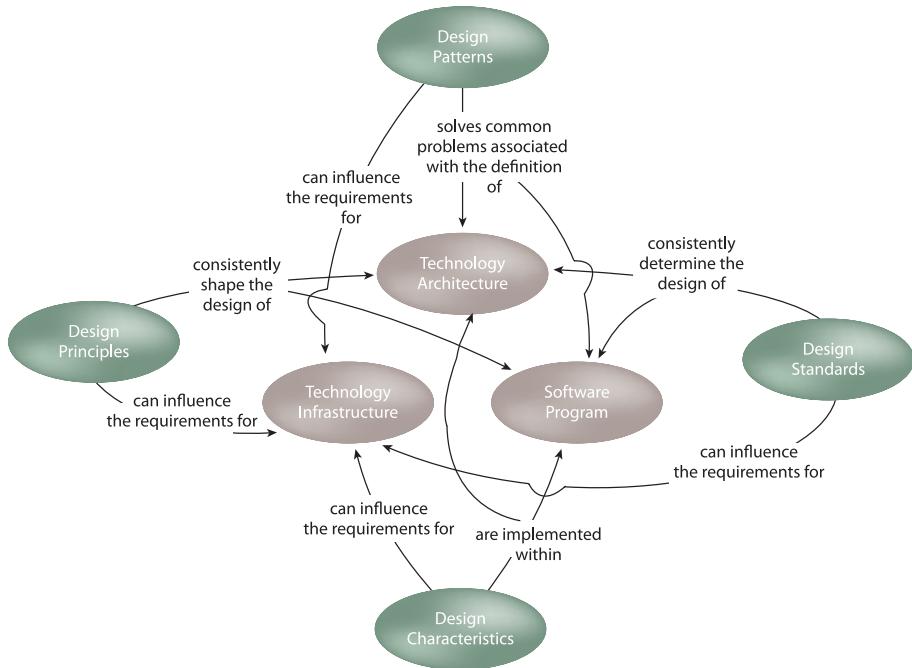


Figure 3.6

How the elements of a design framework can be associated with architecture-related elements.

SUMMARY OF KEY POINTS

- Technology architecture represents the physical structure and aspects of physical design for a piece of technology. Traditional variations of technology architecture include component, application, integration, and enterprise technology architecture. (These will be compared to service-oriented architecture types in Chapter 4.)
 - Technology infrastructure represents the structure and complexion of a technical environment most commonly confined to the boundary of an organization's enterprise. Technology architectures encompass portions of relevant infrastructure.
 - Technology architectures encompass portions of relevant infrastructure.
 - A software program is a system or application that is partially defined by and also resides within a technology architecture.
-

3.2 Service-Oriented Computing Fundamentals

The upcoming sections provide descriptions of common terms used throughout this book.

Service-Oriented Computing

Service-oriented computing is an umbrella term that represents a new generation distributed computing platform. As such, it encompasses many things, including its own design paradigm and design principles, design patterns, a distinct architectural model, and related concepts, technologies, and frameworks.

Service-oriented computing builds upon past distributed computing platforms and adds new design layers, governance considerations, and a vast set of preferred implementation technologies, several of which are based on the Web services framework.

In this book we make reference to the strategic goals of service-oriented computing as they relate to the application of service-orientation and the design of service-oriented architecture. These goals are briefly described in the section *The Method of Service-Orientation* in Chapter 4.

Service-Orientation

Service-orientation is a design paradigm intended for the creation of solution logic units that are individually shaped so that they can be collectively and repeatedly utilized in support of the realization of the specific strategic goals and benefits associated with SOA and service-oriented computing.

Solution logic designed in accordance with service-orientation can be qualified with “service-oriented,” and units of service-oriented solution logic are referred to as “services.” As a design paradigm for distributed computing, service-orientation can be compared to object-orientation (or object-oriented design). Service-orientation, in fact, has many roots in object-orientation and has also been influenced by other industry developments, as shown in Figure 3.7.

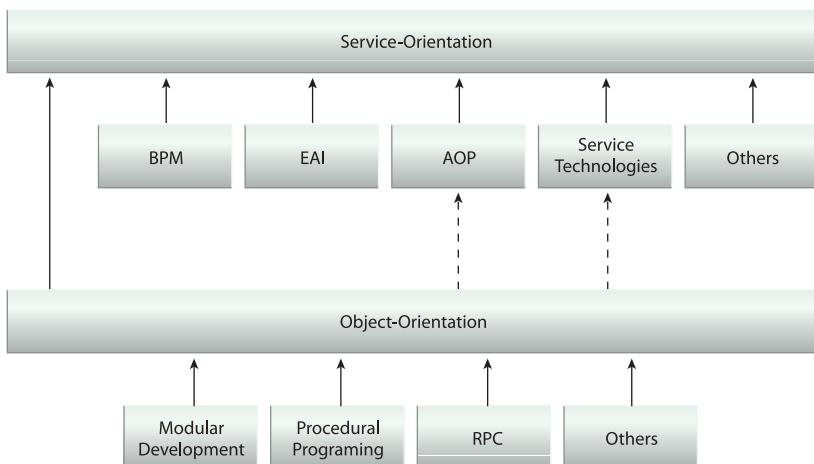


Figure 3.7

Service-orientation is very much an evolutionary design paradigm that owes much of its existence to established design practices and technology platforms.

The service-orientation design paradigm is primarily comprised of eight design principles, which are listed in the section *The Method of Service-Orientation* in Chapter 4. These principles are further summarized in Appendix C and referenced by all subsequent design pattern descriptions so as to highlight potential relationships between the application of a pattern and the application of related design principles.

Service-Oriented Architecture (SOA)

Service-oriented architecture represents an architectural model that aims to enhance the agility and cost-effectiveness of an enterprise while reducing the burden of IT on the overall organization. It accomplishes this by positioning services as the primary means through which solution logic is represented. SOA supports service-orientation in the realization of the strategic goals associated with service-oriented computing. Historically, the term “service-oriented architecture” (or “SOA”) has been used so broadly by the media and within vendor marketing literature that it has almost become synonymous with service-oriented computing itself.

As a form of technology architecture, an SOA implementation can consist of a combination of technologies, products, APIs, supporting infrastructure extensions, and various other parts. The actual complexion of a deployed service-oriented architecture is unique within each enterprise; however it is typified by the introduction of new technologies and platforms that specifically support the creation, execution, and evolution of service-oriented solutions. As a result, building a technology architecture around the service-oriented architectural model establishes an environment suitable for solution logic that has been designed in compliance with service-orientation design principles.

Chapter 4 establishes distinct types of service-oriented architecture and documents four key characteristics that each variation should possess in order to fully support service-orientation. These architecture types are then further referenced by design pattern descriptions in order to highlight the potential scope or applicability of a given pattern.

Service

A *service* is a unit of solution logic (Figure 3.8) to which service-orientation has been applied to a meaningful extent. It is the application of service-orientation design principles that distinguishes a unit of logic as a service compared to units of logic that may exist solely as objects or components.

Subsequent to conceptual service modeling, service-oriented design and development stages implement a service as a physically independent software program with specific design characteristics that support the attainment of the strategic goals associated with service-oriented computing.



Figure 3.8

The chorded circle symbol is used to represent a service, primarily from a contract perspective.

A service corresponds, in scope, to the service architecture type described in Chapter 4. The design patterns in Part III of this book are further dedicated to or related to the design of services.

Service Capability

Each service is assigned its own distinct functional context and is comprised of a set of functions or *capabilities* related to this context. Therefore, a service can be considered a container of capabilities associated with a common purpose (based on a common functional context). The individual bullet items within the service symbol from Figure 3.8 are capabilities.

The term *service capability* has no implication as to how a service is implemented. Therefore, this term can be especially useful during service modeling stages when the physical design of a service has not yet been determined (at which point it is further qualified as a *service capability candidate*). As explained later in the *Service Implementation Mediums* section, once it is known whether a service exists as a Web service, a REST service, or a component, terms such as “service method” or “service operation” can be used instead.

Service Consumer

When a program invokes and interacts with a service it is labeled as a *service consumer*. It is very important to understand that this term refers to the temporary runtime role assumed by a program when it is engaging a service in a data exchange.

For example, you can create a desktop application that is capable of exchanging messages with a service. When it is interacting with the service, the desktop application is considered a service consumer.

You can also design a service to invoke and interact with other services (which, in fact, forms the basis of a service composition, as explained in the next section). In this case, the service itself will temporarily act as the service consumer (Figure 3.9).

Many of the examples in the upcoming chapters show consumers as services. Furthermore, due to the unpredictable nature of data flow through service compositions, the directionality between consumers and services is intentionally not standardized in diagrams provided in this book (which is a deviation from the left-to-right convention used with traditional client-server illustrations).

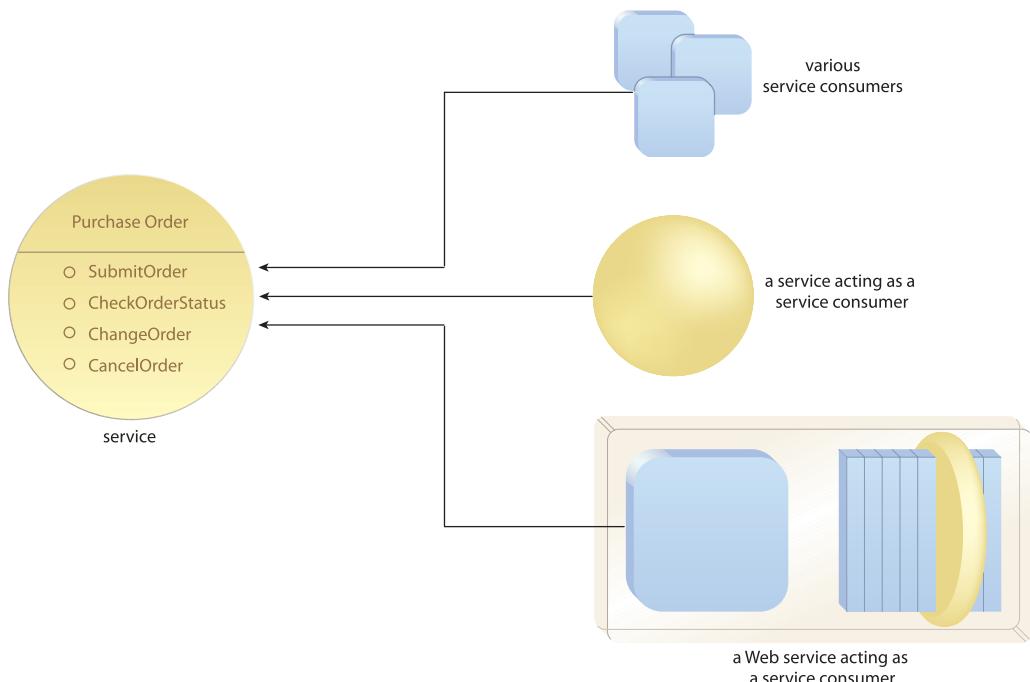


Figure 3.9

The Purchase Order service being accessed by a variety of programs acting as service consumers.

A common alternative term for service consumer is *service requester*. Also, a term often used to represent the runtime role a service assumes when it is being invoked by a service consumer is *service provider*. These terms originated with the W3C and are therefore often used when services are built as Web services, as shown in Figure 3.10.

Many of the patterns in this book deal with design issues that are related to the interaction between service and consumer. Be sure to constantly keep in mind that any service can also be a service consumer.

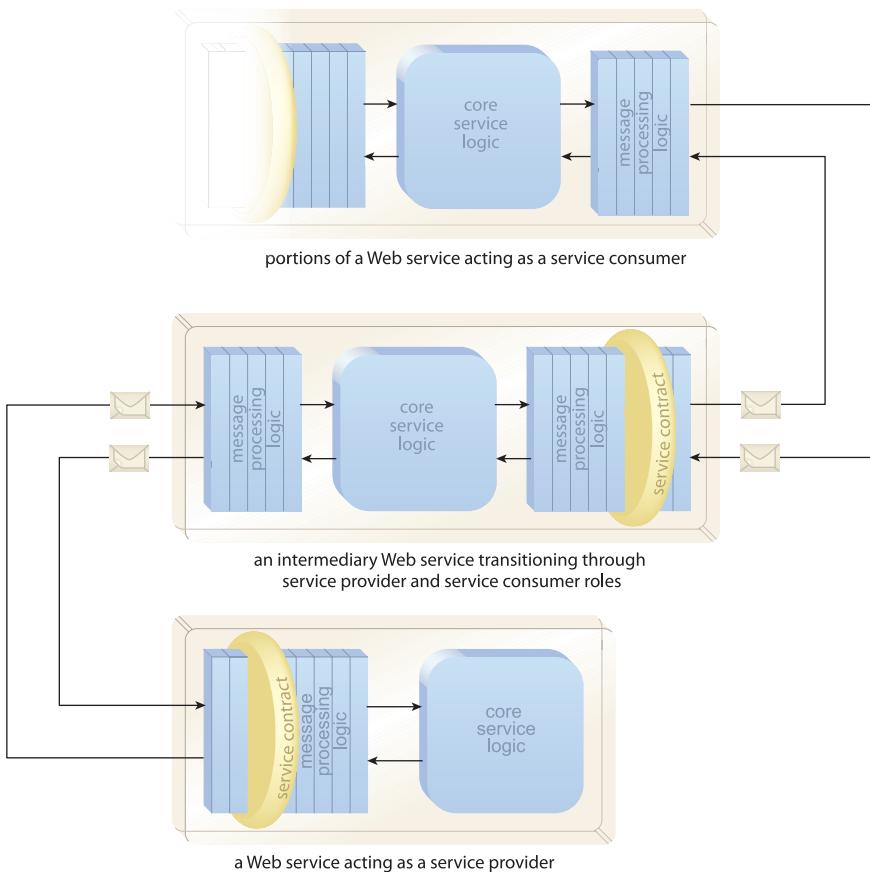


Figure 3.10

Three variations of a Web service showing the different physical parts of its architecture that come into play, depending on the role it assumes at runtime. Note how a service that transitions through service provider and consumer roles is further classified as an intermediary.

Service Composition

A *service composition* is an aggregate of services collectively composed to automate a particular task or business process (Figure 3.11). To qualify as a composition, at least two participating services plus a composition initiator need to be present. Otherwise, the service interaction only represents a point-to-point exchange.

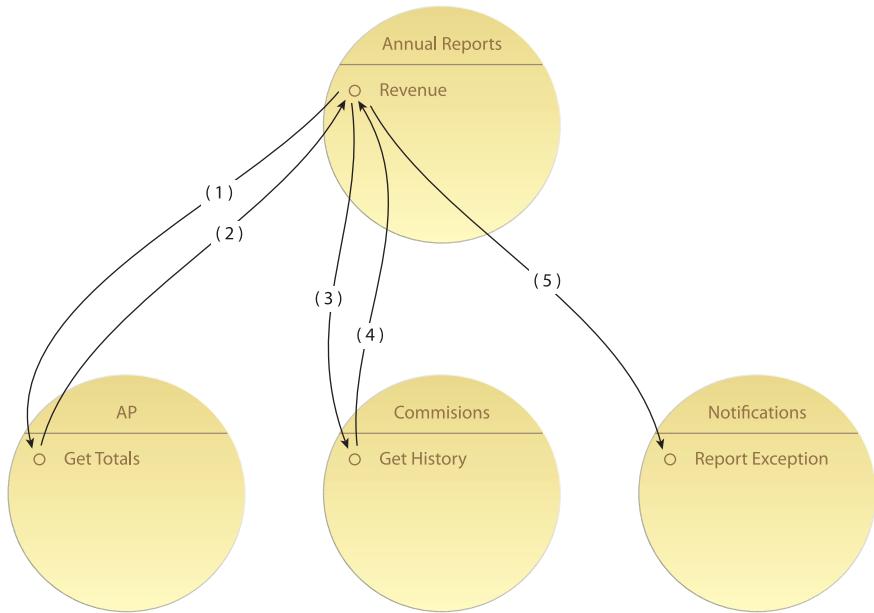


Figure 3.11

A service composition comprised of four services. The arrows indicate a sequence of modeled message exchanges. Note arrow #5 representing a one-way, asynchronous data delivery.

Much of the service-orientation design paradigm revolves around preparing services for effective participation in numerous complex compositions—so much so that the Service Composability design principle is dedicated solely to ensuring that services are designed in support of repeatable composition. A service composition in itself also represents one of the four SOA types explained in Chapter 4.

The design of a composition entails various architectural considerations in order to ensure that runtime service activities can be carried out as expected. Part IV of this book provides a set of chapters with patterns focused on composition design and related runtime processing.

NOTE

There are several additional terms associated with service composition design, including:

- composition controller
- composition controller capability
- composition initiator
- composition member
- composition member capability
- composition sub-controller
- service activity

Definitions for these terms are available at SOAGlossary.com.

Service Inventory

A *service inventory* is an independently standardized and governed collection of complementary services within a boundary that represents an enterprise or a meaningful segment of an enterprise (Figure 3.12). When an organization has multiple service inventories, this term is further qualified as *domain service inventory*, as explained in the pattern description for Domain Inventory (123).

Service inventories are typically created through top-down delivery processes that result in the definition of *service inventory blueprints*. The subsequent application of service-orientation design principles and custom design standards throughout a service inventory is of paramount importance so as to establish a high degree of native inter-service interoperability. This supports the repeated creation of effective service compositions in response to new and changing business requirements.

The service inventory architecture is one of four SOA types explained in the following chapter. Part II of this book further provides a collection of fundamental and specialized patterns dedicated to service inventory design.

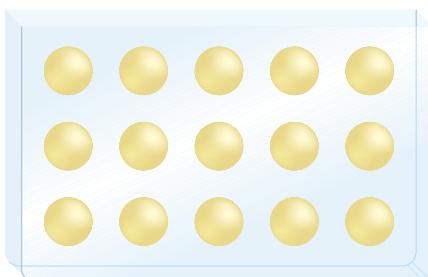


Figure 3.12

The standard symbol used to represent a service inventory in this book is the open blue container.

Where Did the Term “Service Inventory” Come From?

“When building services as part of an SOA project, there is an emphasis on developing them as standalone programs that are expected to be flexible and robust so that they can be readily reused and composed. Service-oriented design has therefore been heavily influenced by commercial product design techniques, to the extent that a service delivered as a black box is somewhat comparable to a software product. Inspired by commercial terminology, a collection of services for a given segment of an enterprise is referred to as a *service inventory*. And, similarly, the technology architecture that supports this collection of services is referred to as the *service inventory architecture*.

What’s the difference between a service inventory and a service catalog? The same manner in which an inventory of products is documented with a product catalog, an inventory of services is documented with a service catalog. It’s therefore still appropriate to refer to a collection of services as a service catalog; however, when applying design patterns and defining the actual concrete architecture, terms like “service inventory” (or even “service pools”) tend to work better.”

– “Introducing SOA Design Patterns,” SOA World Magazine, June 2008

Service-Oriented Analysis

Service-oriented analysis represents one of the early stages in an SOA initiative and the first phase in the service delivery cycle. It is a process that often begins with preparatory information gathering steps that are completed in support of a service modeling sub-process that results in the creation of conceptual service candidates, service capability candidates, and service composition candidates.

The service-oriented analysis process is commonly carried out iteratively, once for each business process. When applied as part of a top-down approach, the scope of a planned service inventory will generally determine the extent of the service-oriented analysis effort. All iterations of a service-oriented analysis then pertain to that scope, with the goal of producing a service inventory blueprint. (Visit SOAMethodology.com for an explanation of the iterative service-oriented analysis process.)

A key success factor of service-oriented analysis is the hands-on collaboration of both business analysts and technology architects. The former group is especially involved in the definition of service candidates with a business-centric functional context because they understand the business processes used as input for the analysis and because service-orientation aims to align business and IT more closely.

The service definition process that is commonly carried out as part of a service-oriented analysis will usually contain steps that correspond to the foundational service patterns provided in Chapter 11.

Service Candidate

When conceptualizing services during the service modeling part of the service-oriented analysis phase, services are defined on a preliminary basis and still subject to change and refinement before they are handed over to the service-oriented design project stage responsible for producing the physical service architecture. The term *service candidate* is used to help distinguish a conceptualized service from an actual implemented service.

3.3 Service Implementation Mediums

It is important to view and position SOA as an architectural model that is neutral to any one technology platform. By doing so, an enterprise is given the freedom to continually pursue the strategic goals associated with SOA and service-orientation by leveraging on-going technology advancements.

Currently, a service can be built and implemented as a:

- component
- Web service
- REST service

Essentially, any implementation technology that can be used to create a distributed system may be suitable for service-orientation.

Many of the design patterns in this book are not specific to any one of these three implementation mediums, but some are. For example, several examples in this book are based on the use of Web services because this service implementation medium has been historically the most popular.

The remaining sections in this chapter briefly introduce each of these implementation options. However, because this book is dedicated to design patterns, complete descriptions of these technologies are intentionally deferred to other series titles.

Services as Components

A *component* is a software program designed to be part of a distributed system. It provides a technical interface comparable to a traditional application programming interface (API) through which it exposes public capabilities as *methods*, thereby allowing it to be explicitly invoked by other programs (Figure 3.13).

Components typically rely on platform-specific development and runtime technologies. For example, components can be built using Java or .NET tools and are then deployed in a runtime environment capable of supporting the corresponding component communications technology requirements, as implemented by the chosen development platform.

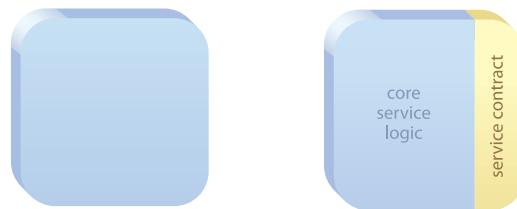


Figure 3.13

The symbols used to represent a component. The symbol on the left is a generic component that may or may not have been designed as a service, whereas the symbol on the right is explicitly labeled to indicate that it has been designed as a service.

NOTE

Building service-oriented components is one of the topics covered in the upcoming books *SOA with Java* and *SOA with .NET*.

Services as Web Services

A *Web service* is a body of solution logic that provides a physically decoupled technical contract consisting of a WSDL definition and one or more XML Schema definitions and also possible WS-Policy expressions. The Web service contract exposes public capabilities as *operations*, establishing a technical interface but without any ties to a proprietary communications framework (Figure 3.14).

Service-orientation can be applied to the design of Web services. The fact that Web services provide an architectural model whereby the service contract is physically decoupled and vendor-neutral is conducive to several of the design goals associated with service-orientation.

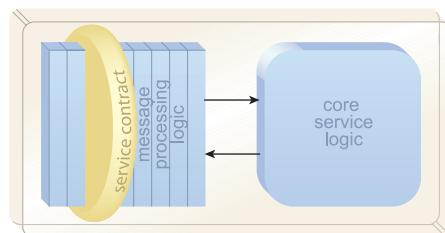


Figure 3.14

The typical Web service architecture containing a service contract, a component, and message processing logic comprised of event-driven agents, as per the pattern Service Agent (543).

NOTE

Coverage of Web services in relation to SOA is provided by the books *Web Service Contract Design and Versioning for SOA* and *Service-Oriented Architecture: Concepts, Technology, and Design*.

REST Services

Representational State Transfer (REST) provides a means of constructing distributed systems based on the notion of *resources*. REST services (or RESTful Services) are lightweight programs that are designed with an emphasis on simplicity, scalability, and usability. REST services can be further shaped by the application of service-orientation principles.

As previously explained in the *Web Service and REST Service Design Patterns* section in Chapter 1, several REST-inspired candidate design patterns have been developed for this pattern catalog and are published at SOAPatterns.org.

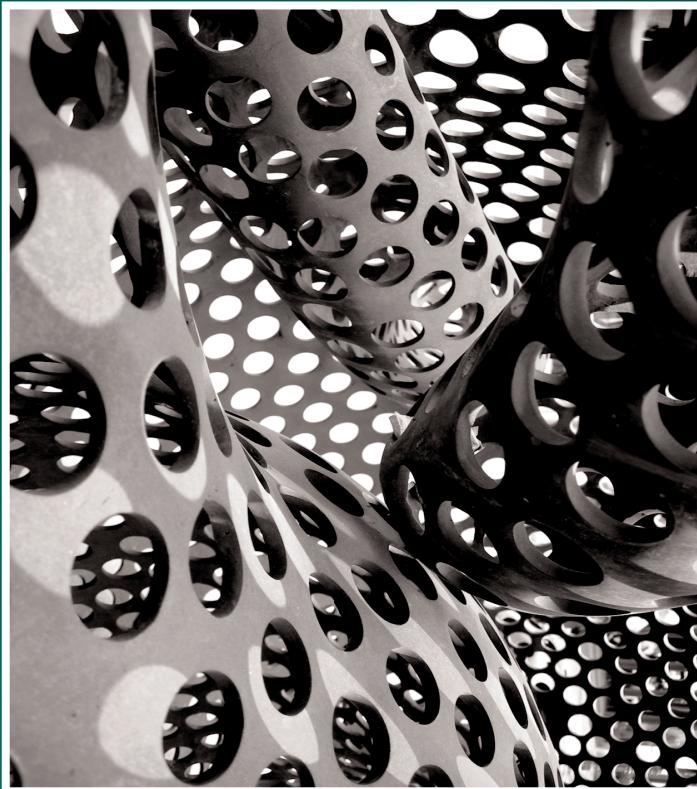
NOTE

How REST services can be built in support of SOA and service-orientation is explored in the upcoming series titles *SOA with REST*, *SOA with Java*, and *SOA with .NET*.

SUMMARY OF KEY POINTS

- Service-orientation is a design paradigm that can be applied to any suitable distributed computing technology platform.
 - There are currently three common implementation mediums suitable for building services: components, Web services, and REST services.
-

Chapter 4



The Architecture of Service-Orientation

- 4.1 The Method of Service-Orientation
- 4.2 The Four Characteristics of SOA
- 4.3 The Four Common Types of SOA
- 4.4 The End Result of Service-Orientation

Service-oriented computing is fundamentally about attaining a specific target state. It asks that we take extra design considerations into account with everything we build so that all the moving parts of a service-oriented solution support the realization of this state and foster its growth and evolution. This target state is attractive because it has associated with it a specific set of goals and benefits.

To fully understand service-oriented technology architecture requires knowledge of:

- how these goals and benefits are achieved (the method)
- what entails the attainment of these goals and benefits (the end-result)

This understanding allows us to assess what requirements and demands are placed upon technology architecture.

Purpose of this Introductory Chapter

The focus of this chapter is on establishing the relationship between service-orientation and service-oriented architecture by highlighting common architectural characteristics required to support the goals of service-orientation. This chapter furthermore documents types of service-oriented architecture that are referenced later in design pattern descriptions.

4.1 The Method of Service-Orientation

To realize the strategic benefits of service-oriented computing requires that each piece of solution logic be designed consistently and in a manner that fully supports the expected target environment. This is the role of service-orientation. It is the fundamental method by which service-oriented solutions are created.

Principles of Service-Orientation

There are eight distinct design principles that are part of the service-orientation design paradigm. Each addresses a key aspect of service design by ensuring that specific design characteristics are consistently realized within every service. When collectively applied to a

meaningful extent, service-orientation design principles shape solution logic into something we can legitimately refer to as “service-oriented.”

Below are the eight service-orientation design principles together with their official definitions:

- *Standardized Service Contract* – Services within the same service inventory are in compliance with the same contract design standards.
- *Service Loose Coupling* – Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment.
- *Service Abstraction* – Service contracts only contain essential information and information about services is limited to what is published in service contracts.
- *Service Reusability* – Services contain and express agnostic logic and can be positioned as reusable enterprise resources.
- *Service Autonomy* – Services exercise a high level of control over their underlying runtime execution environment.
- *Service Statelessness* – Services minimize resource consumption by deferring the management of state information when necessary.
- *Service Discoverability* – Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.
- *Service Composability* – Services are effective composition participants, regardless of the size and complexity of the composition.

Figure 4.1 provides some perspective as to how these principles affect the design of a service. The application of the principles on the right side tend to result in concrete design characteristics being added to a service, whereas the principles on the left usually act as regulatory influences, ensuring a balanced application of service-orientation as a whole.

As just mentioned, a solution is considered service-oriented once service-orientation has been applied to a meaningful extent. A mere understanding of the design paradigm, however, is insufficient. To apply service-orientation consistently and successfully requires a technology architecture customized to accommodate its design preferences, initially when services are first delivered and especially when collections of services are accumulated and assembled into complex compositions.

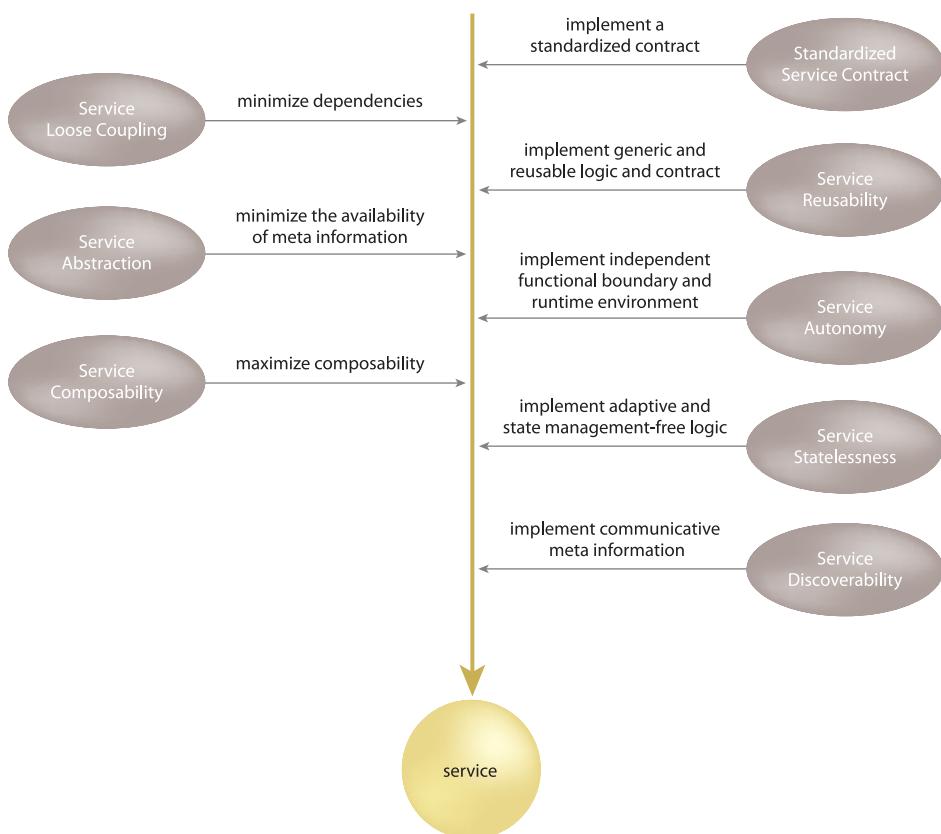


Figure 4.1

How service-orientation design principles relate to each other and how they collectively shape service design.

NOTE

Design principles are referenced throughout this book but represent a separate subject-matter that is covered in *SOA Principles of Service Design*. Introductory coverage of service-orientation is also available at SOAPrinciples.com.

Strategic Goals of Service-Oriented Computing

Service-orientation emerged as a design approach in support of achieving the following goals and benefits associated with SOA and service-oriented computing:

- *Increased Intrinsic Interoperability* – Services within a given boundary are designed to be naturally compatible so that they can be effectively assembled and reconfigured in response to changing business requirements.
- *Increased Federation* – Services establish a uniform contract layer that hides underlying disparity, allowing them to be individually governed and evolved.
- *Increased Vendor Diversification Options* – A service-oriented environment is based on a vendor-neutral architectural model, allowing the organization to evolve the architecture in tandem with the business without being limited to proprietary vendor platform characteristics.
- *Increased Business and Technology Domain Alignment* – Some services are designed with a business-centric functional context, allowing them to mirror and evolve with the business of the organization.
- *Increased ROI* – Most services are delivered and viewed as IT assets that are expected to provide repeated value that surpasses the cost of delivery and ownership.
- *Increased Organizational Agility* – New and changing business requirements can be fulfilled more rapidly by establishing an environment in which solutions can be assembled or augmented with reduced effort by leveraging the reusability and native interoperability of existing services.
- *Reduced IT Burden* – The enterprise as a whole is streamlined as a result of the previously described goals and benefits, allowing IT itself to better support the organization by providing more value with less cost and less overall burden.

NOTE

Formal descriptions for these strategic goals are available at WhatIsSOA.com and in Chapter 3 of *SOA Principles of Service Design*.

When studying the design patterns in this book that support service-orientation, it is important to keep these goals (and the target state they represent) in mind. Understanding the ultimate state attainable provides us with a constant strategic context for each pattern. In other words, it helps provide insight into why certain parts of a service-oriented environment need to be designed in certain ways, which may not always be evident when reading through the pattern descriptions individually.

SUMMARY OF KEY POINTS

- A key ingredient to attaining the strategic goals and benefits associated with service-oriented computing is the successful application of the service-orientation design paradigm.
 - Service-oriented architecture needs to be designed in support of service-orientation in order to support the realization of these strategic goals and benefits.
 - Understanding the strategic goals helps clarify the design solutions proposed by SOA design patterns.
-

4.2 The Four Characteristics of SOA

Having just explained the service-orientation design paradigm and its associated goals, we now need to turn our attention to the physical design of a service-oriented solution or environment.

In support of achieving the goals of service-orientation, there are four base characteristics we look to establish in any form of SOA:

- *Business-Driven* – The technology architecture is aligned with the current business architecture. This context is then constantly maintained so that the technology architecture evolves in tandem with the business over time.
- *Vendor-Neutral* – The architectural model is not based solely on a proprietary vendor platform, allowing different vendor technologies to be combined or replaced over time in order to maximize business requirements fulfillment on an on-going basis.

- *Enterprise-Centric* – The scope of the architecture represents a meaningful segment of the enterprise, allowing for the reuse and composition of services and enabling service-oriented solutions to span traditional application silos.
- *Composition-Centric* – The architecture inherently supports the mechanics of repeated service aggregation, allowing it to accommodate constant change via the agile assembly of service compositions.

These characteristics help distinguish SOA from other architectural models and also define the fundamental requirements a technology architecture must fulfill to be fully supportive of service-orientation. As we explore each individually, keep in mind that in real-world implementations the extent to which these characteristics can be attained tends to vary.

Business-Driven

Traditional technology architectures were commonly designed in support of solutions delivered to fulfill tactical (short-term) business requirements. Because the overarching, strategic (long-term) business goals of the organization aren't taken into consideration when the architecture is defined, this approach can result in a technical environment that, over time, becomes out of alignment with the organization's business direction and requirements (Figure 4.2).

This gradual separation of business and technology results in a technology architecture with diminishing potential to fulfill business requirements and one that is increasingly difficult to adapt to changing business needs.

When a technology architecture is business-driven, the overarching business vision, goals, and requirements are positioned as the basis for and the primary influence of the architectural model. This maximizes the potential alignment of technology and business and allows for a technology architecture that can evolve in tandem with the organization as a whole (Figure 4.3). The result is a continual increase in the value and lifespan of the architecture.

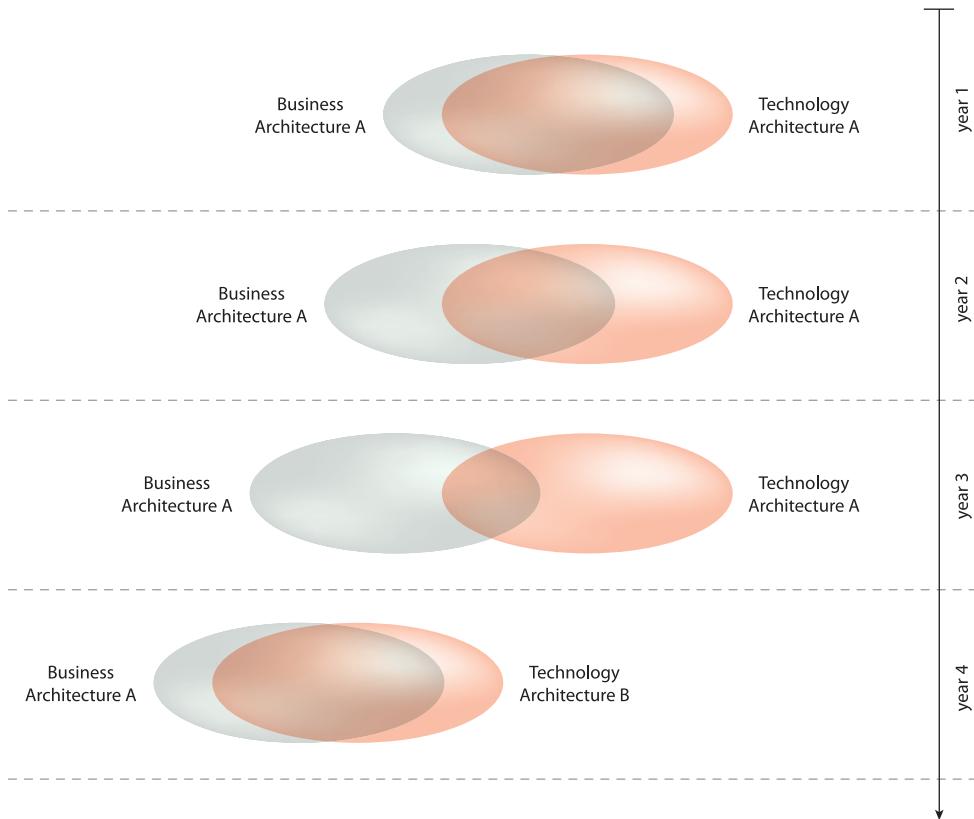


Figure 4.2

A traditional technology architecture (A) is often delivered in alignment with the current state of a business but can be incapable of changing in alignment with how the business evolves. As business and technology architectures become increasingly out of sync, business requirement fulfillment decreases, often to the point that a whole new technology architecture (B) is needed, which effectively resets this cycle.

Vendor-Neutral

Designing a service-oriented technology architecture around one particular vendor platform can lead to an implementation that inadvertently inherits proprietary characteristics. This can end up inhibiting the future evolution of an inventory architecture in response to technology innovations that become available from other vendors.

An inhibitive technology architecture is unable to evolve and expand in response to changing automation requirements, which can result in the architecture having a limited lifespan after which it needs to be replaced to remain effective (Figure 4.4).

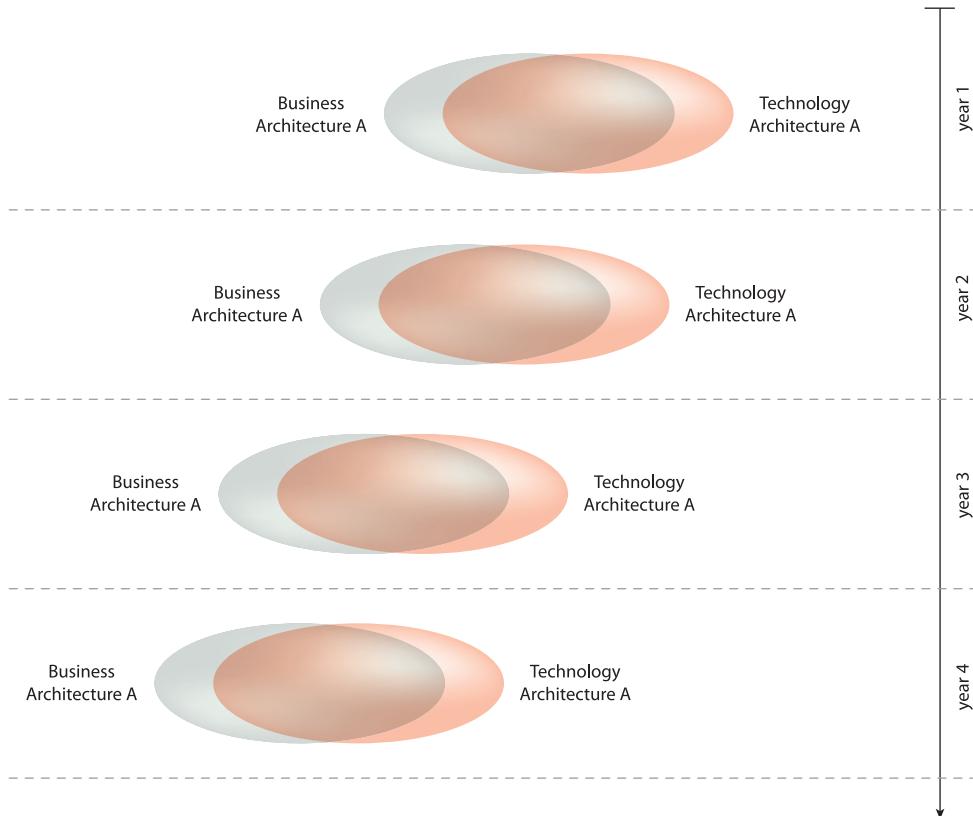


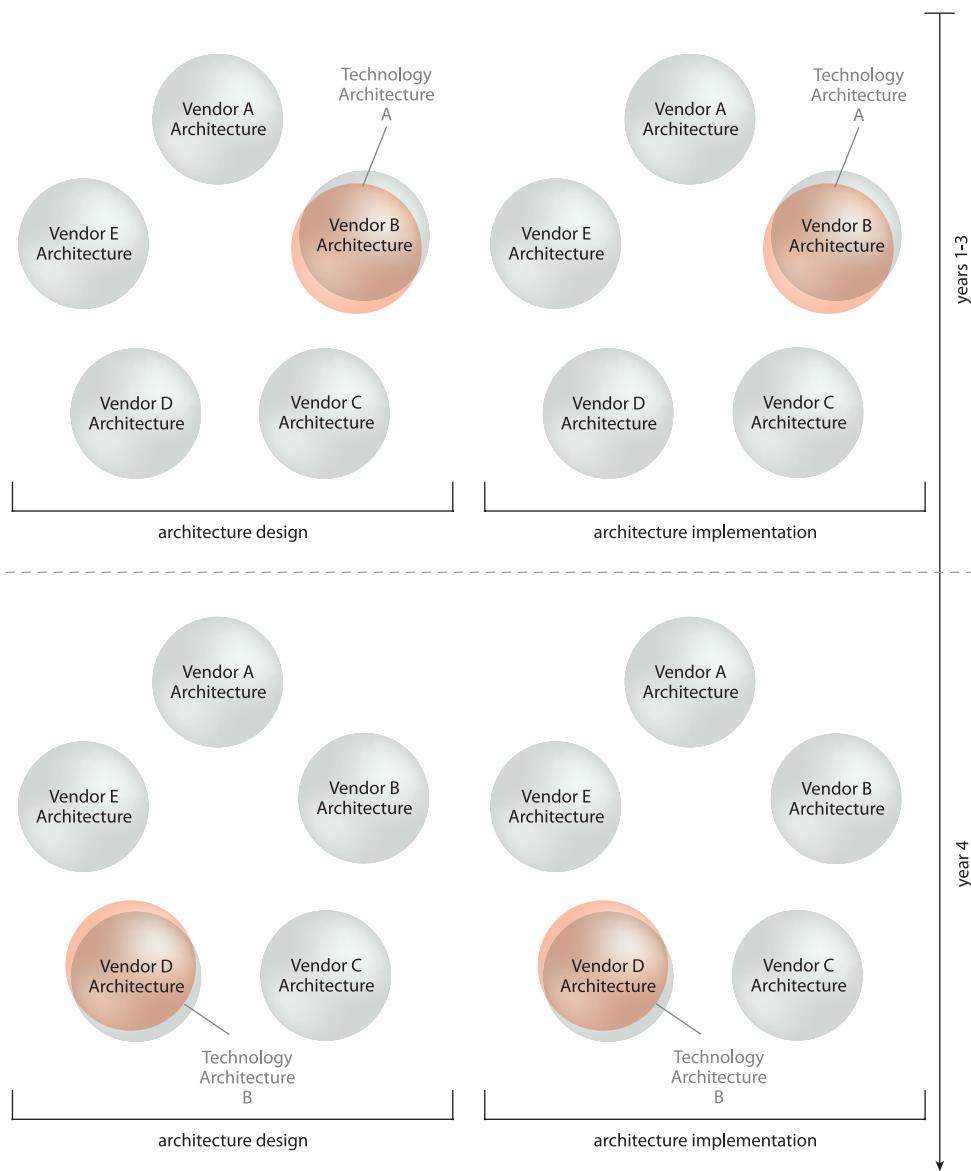
Figure 4.3

By defining a strategic, business-centric scope to the technology architecture, it can be kept in constant sync with how the business evolves over time.

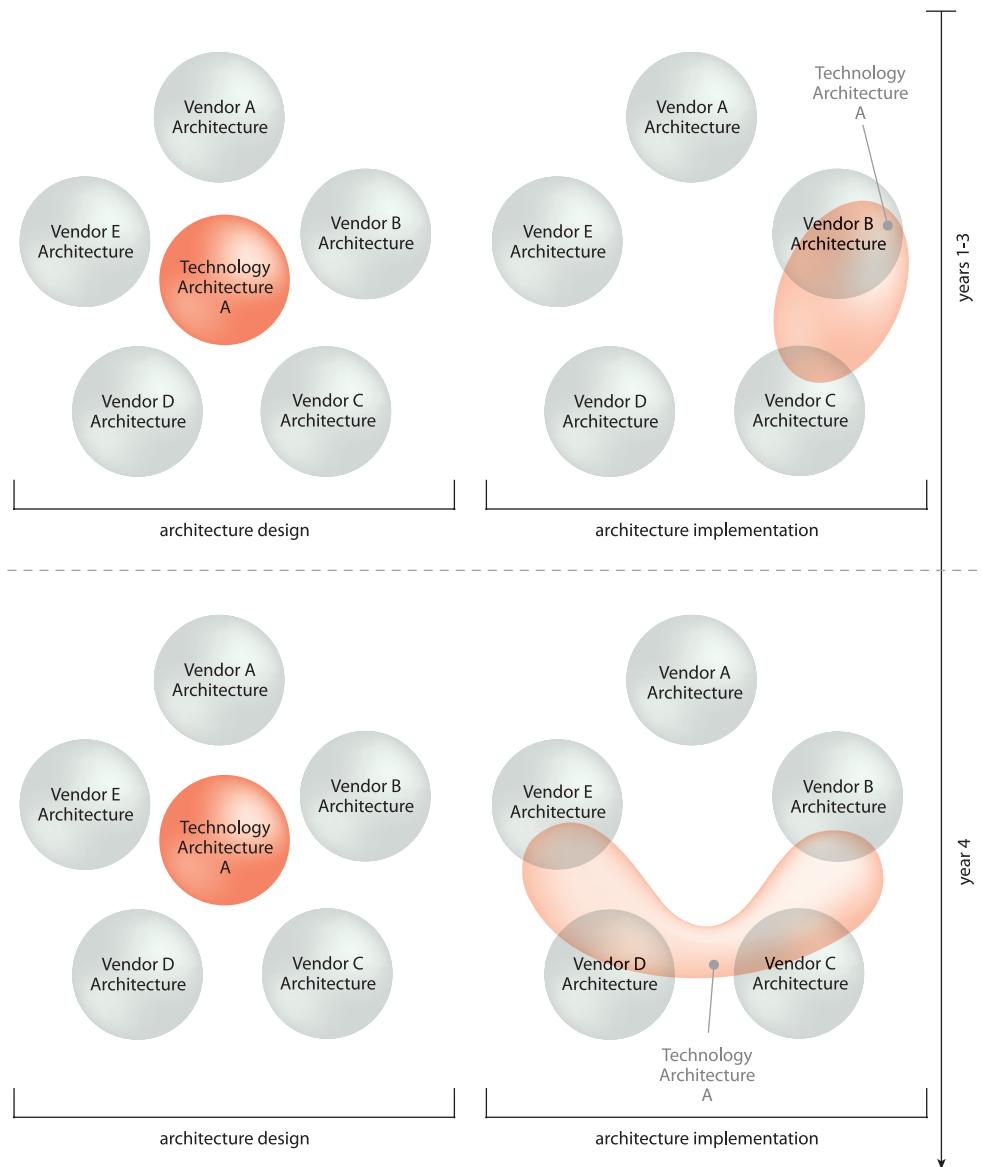
It is in the best interest of an organization to base the design of a service-oriented architecture on a model that is in alignment with the primary SOA vendor platforms, yet neutral to all of them. A vendor-neutral architectural model can be derived from a vendor-neutral design paradigm used to build the solution logic the architecture will be responsible for supporting. The service-orientation paradigm provides such an approach, in that it is derived from and applicable to real world technology platforms while remaining independent of them.

NOTE

Just because an architecture is classified as vendor-neutral doesn't mean it is also *aligned* with current vendor technology. Some models produced by independent efforts are out of sync with the manner in which mainstream SOA technology exists today and is expected to evolve in the future and can therefore be just as inhibitive as vendor-specific models.

**Figure 4.4**

Vendor-centric technology architectures are often bound to corresponding vendor platform roadmaps. This can reduce opportunities to leverage technology innovations provided by other vendor platforms and can result in the need to eventually replace the architecture entirely with a new vendor implementation (which starts the cycle over again).

**Figure 4.5**

If the architectural model is designed to be and remain neutral to vendor platforms, it maintains the freedom to diversify its implementation by leveraging multiple vendor technology innovations. This increases the longevity of the architecture as it is allowed to augment and evolve in response to changing requirements.

Enterprise-Centric

Just because solutions are based on a distributed architecture doesn't mean that there still isn't the constant danger of creating new silos. In fact, it has been common to build distributed solutions comprised of single-purpose components. Software programs labeled as services that are designed in this manner naturally result in silos (Figure 4.6) that continue to bloat the enterprise and lead to traditional integration requirements.

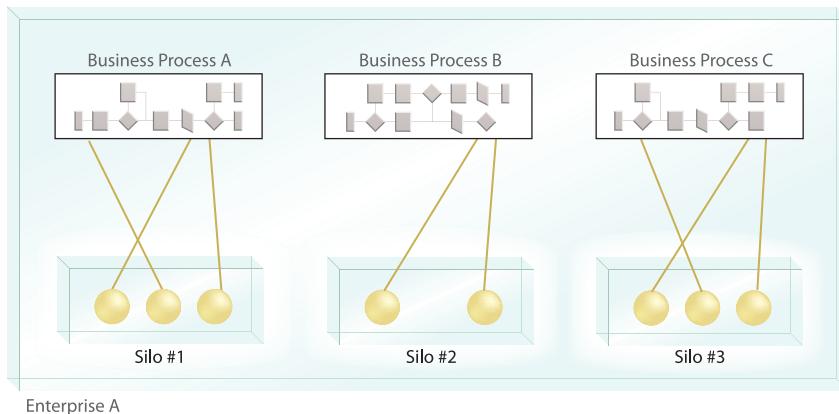


Figure 4.6

Single-purpose services delivered to automate specific business processes end up establishing silos within the enterprise.

When applying service-orientation, services are positioned as *enterprise resources*, which implies that service logic is designed with the following primary characteristics:

- The logic is available beyond a specific implementation boundary.
- The logic is designed according to established design principles and enterprise standards.

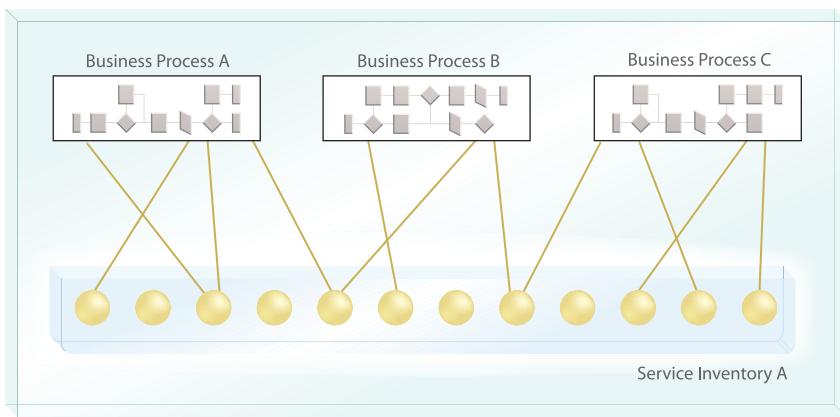
Essentially, the body of logic is classified as a resource of the enterprise. This does not necessarily make it an enterprise-wide resource or one that must be used throughout an entire technical environment. In other words, an enterprise resource does not belong solely to any one application or solution environment.

As further established in the pattern description for Service Encapsulation (305), an enterprise resource essentially embodies the fundamental characteristics of service logic.

NOTE

See the *Enterprise vs. Enterprise-wide* section in Chapter 5 for a comparison of how these two terms are used in this book.

In order to leverage services as enterprise resources, the underlying technology architecture must establish a model that is natively based on the assumption that software programs delivered as services will be shared by other parts of the enterprise or will be part of larger solutions that include shared services. This baseline requirement places an emphasis on standardizing parts of the architecture (as per the canonical patterns introduced in Chapter 5) so that service reuse and interoperability can be continually fostered (Figure 4.7).



Enterprise A

Figure 4.7

When services are positioned as enterprise resources they no longer create or reside in silos. Instead they are made available to a broader scope of utilization by being part of a service inventory.

Composition-Centric

More so than in previous distributed computing paradigms, service-orientation places an emphasis on designing software programs as not just reusable resources, but as flexible resources that can be plugged into different aggregate structures as part of a variety of service-oriented solutions.

To accomplish this, services must be composable. As advocated by the Service Composability principle, this means that services must be capable of being pulled into a variety of composition designs, regardless of whether they are initially required to participate in a composition when they are first delivered (Figure 4.8).

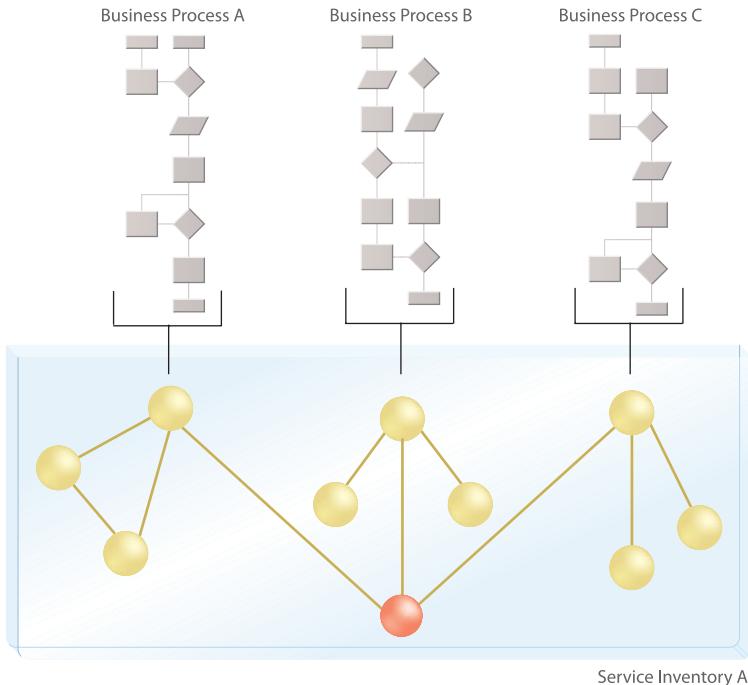


Figure 4.8

Services within the same service inventory are composed into different configurations. The highlighted service is reused by multiple compositions to automate different business processes.

To support native composability, the underlying technology architecture must be prepared to enable a range of simple and complex composition designs. Architectural extensions (and related infrastructure extensions) pertaining to scalability, reliability, and runtime data exchange processing and integrity are essential to support this key characteristic.

NOTE

In the next section we introduce four specific types of service-oriented architecture. The aforementioned architectural characteristics are fundamental to all of these SOA types; however, it is worth singling out the service inventory as the primary starting point for their implementation. We revisit these four characteristics at the beginning of Chapter 6 to highlight how they relate to fundamental inventory patterns.

SUMMARY OF KEY POINTS

- The four fundamental characteristics that any form of SOA needs to have in support of service-orientation are Business-Driven, Vendor-Neutral, Enterprise-Centric, and Composition-Centric.
 - Whereas the Business-Driven and Vendor-Neutral characteristics help shape the overall context and model of a service-oriented architecture, the Enterprise-Centric and Composition-Centric characteristics place demands on the actual technology and infrastructure extensions that the architecture is based upon.
 - The fundamental inventory patterns in Chapter 6 are closely related to these four characteristics.
-

4.3 The Four Common Types of SOA

As we've already established, every software program ends up being comprised of and residing in some form of architectural combination of resources, technologies, and platforms (infrastructure-related or otherwise). If we take the time to customize these architectural elements, we can establish a refined and standardized environment for the implementation of (also customized) software programs.

The intentional design of technology architecture is very important to service-oriented computing. It is essential to establishing an environment within which services can be repeatedly recomposed to maximize business requirements fulfillment. As evidenced by the range of architectural design patterns in this book, the strategic benefit to customizing the scope, context, and boundary of an architecture is significant.

To better understand the basic mechanics of SOA, we now need to study the common types of technology architectures that exist:

- *Service Architecture* – The architecture of a single service.
- *Service Composition Architecture* – The architecture of a set of services assembled into a service composition.
- *Service Inventory Architecture* – The architecture that supports a collection of related services that are independently standardized and governed.
- *Service-Oriented Enterprise Architecture* – The architecture of the enterprise itself, to whatever extent it is service-oriented.

Although they are roughly comparable to the four traditional architecture types described in the *Architecture Fundamentals* section in Chapter 3, each is distinct due to the unique requirements and dynamics of service-orientation (Figure 4.9). The next four sections explore these SOA types individually.

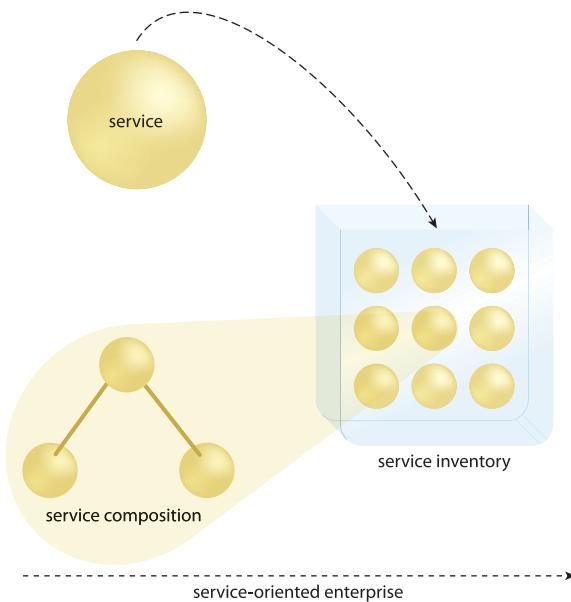


Figure 4.9

Services (top left) are delivered into a service inventory (right) from which service compositions (bottom left) are drawn. These basic elements establish the fundamental service-orientation dynamic but also represent the three common SOA types that can reside within a service-oriented enterprise (bottom).

Service Architecture

A technology architecture limited to the physical design of a software program designed as a service is referred to as the *service architecture*. This form of technology architecture is comparable in scope to a component architecture, except that it will typically rely on a greater amount of infrastructure extensions to support its need for increased reliability, performance, scalability, behavioral predictability, and especially its need for increased autonomy. The scope of a service architecture will also tend to be larger because a service can, among other things, encompass multiple components.

Whereas it was not always that common to document a separate architecture for a component in traditional distributed applications, the importance of producing services that need to exist as independent and highly self-sufficient and self-contained software programs requires that each be individually designed. Figure 4.10 shows a highly simplified view of a sample service architecture.

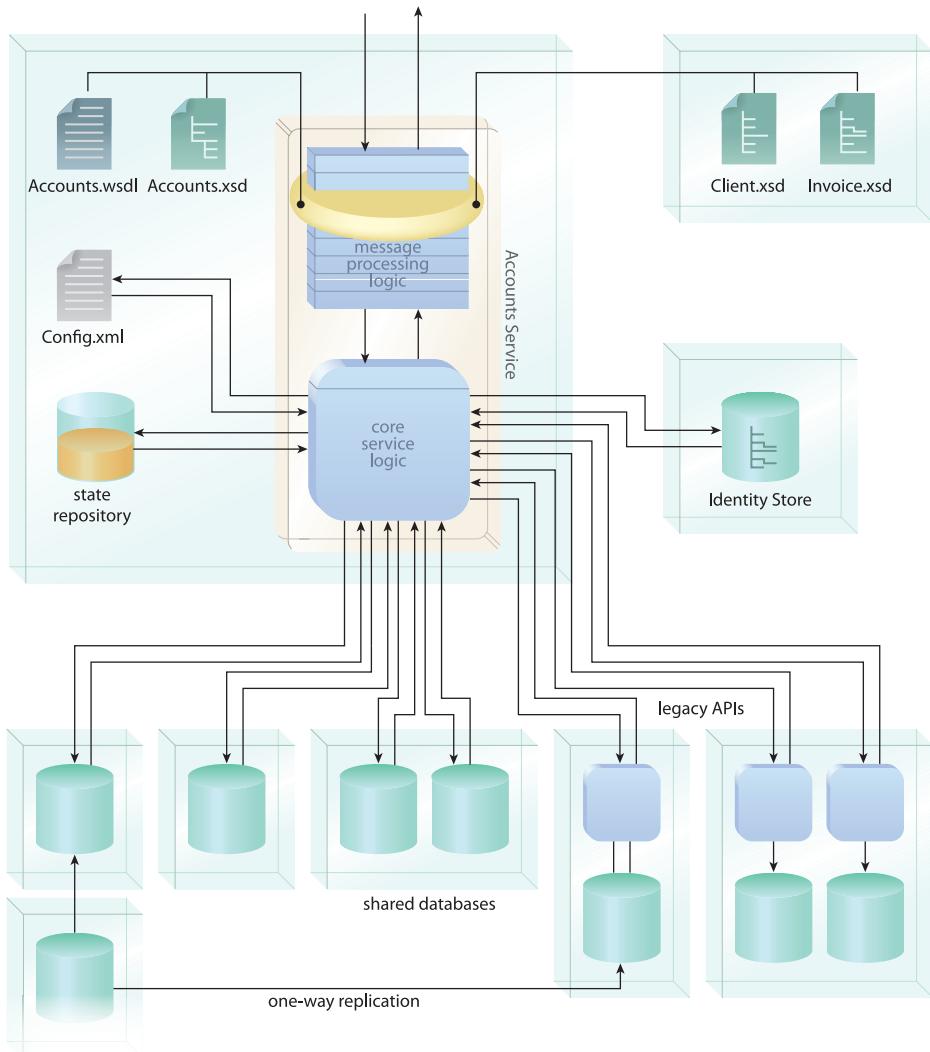


Figure 4.10

An example of a high-level service architecture view for the Accounts Web service, depicting the parts of the surrounding infrastructure utilized to fulfill the functional requirements of all capabilities (or operations). Additional views can be created to show only those architectural elements related to the processing of specific capabilities. Further detail, such as data flow and security requirements, would normally also be included.

Information Hiding

Service architecture specifications are typically owned by service custodians and, in support of the Service Abstraction design principle, their contents are often protected and hidden from other project team members (Figure 4.11).

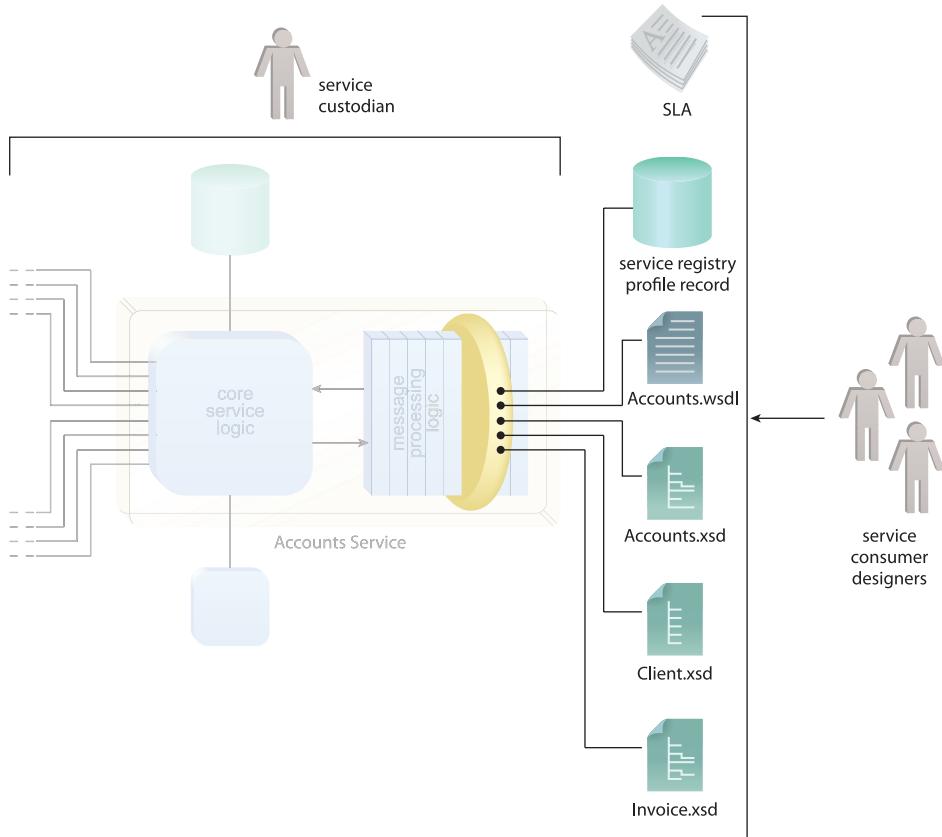


Figure 4.11

The custodian of the Accounts service intentionally limits access to architecture documentation. As a result, service consumer designers are only privy to published service contract documents.

Design Standards

The application of design standards and other service-orientation design principles (Figure 4.12) further affects the depth and detail to which a service's technology architecture may need to be defined. For example, implementation consideration raised by the Service Autonomy and Service Statelessness principles can require a service architecture to extend deeply into its surrounding infrastructure by defining exactly what physical environment it is deployed within, what resources it needs to access, what other parts of the enterprise may

be accessing those same resources, and what extensions from the infrastructure it can use to defer or store data it is responsible for processing.

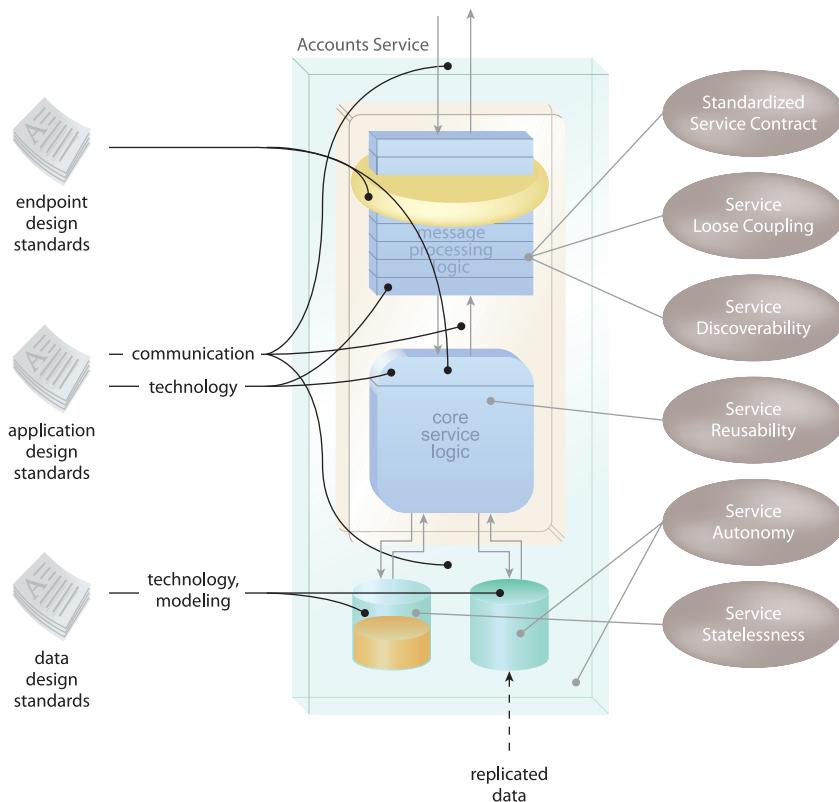


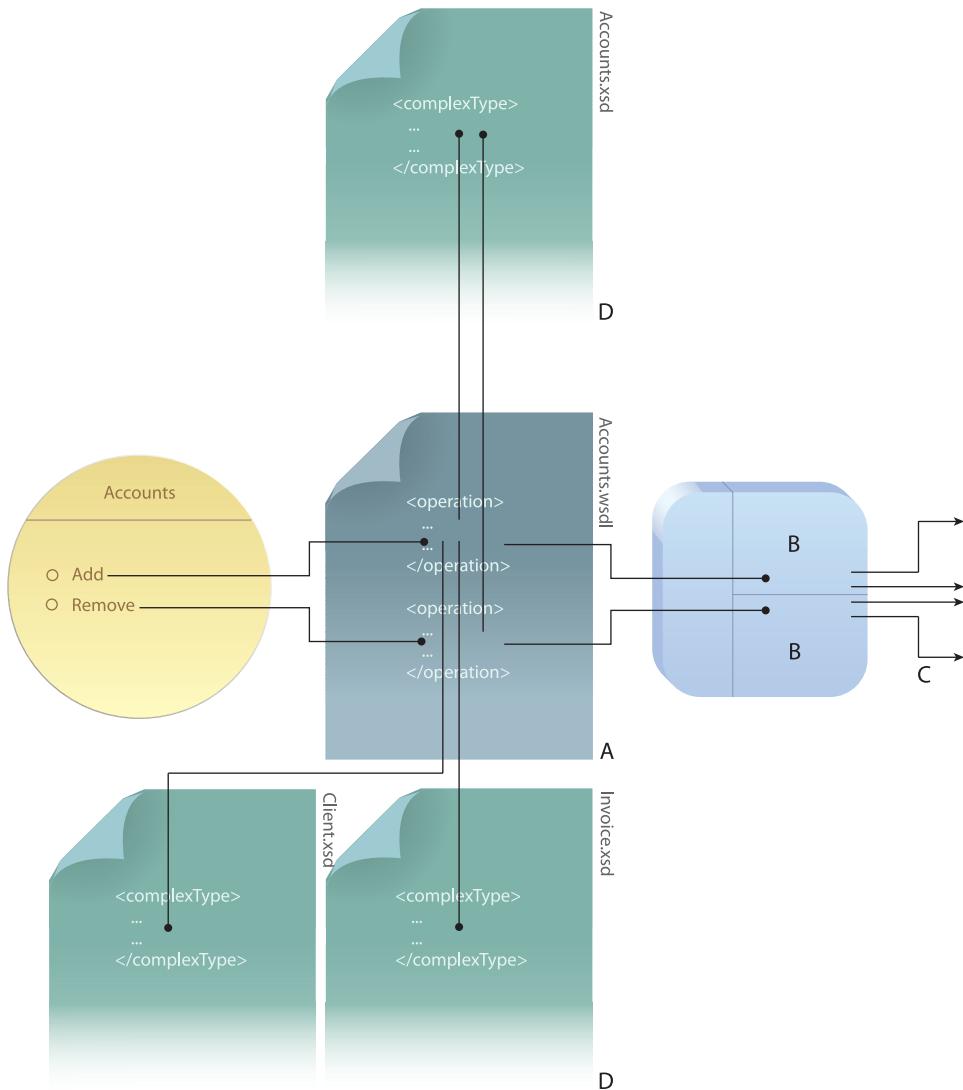
Figure 4.12

Custom design standards and service-orientation design principles are applied to establish a specific set of design characteristics within the Accounts service architecture.

Service Contracts

A central part of a service architecture is typically its technical contract (Figure 4.13). Following standard service-oriented design processes, the service contract is usually the first part of a service to be physically delivered. The capabilities expressed by the contract further dictate the scope and nature of its underlying logic and the processing requirements that will need to be supported by its implementation.

This is why some consideration is given to implementation during a service's modeling phase (which occurs prior to its physical design).

**Figure 4.13**

The service contract is a fundamental part of the Accounts service architecture. Its definition gives the service a public identity and helps express its functional scope. Specifically, the WSDL document (A) expresses operations that correspond to segments of functionality (B) within the underlying Accounts service logic. The logic, in turn, accesses other resources in the enterprise to carry out those functions (C). To accomplish this, the WSDL document provides data exchange definitions via input and output message types established in separate XML Schema documents (D).

NOTE

Many organizations use standard *service profile documents* to collect and maintain information about a service throughout its lifespan. Chapter 15 of *SOA Principles of Service Design* explains the service profile document and provides a sample template.

Service Agents

Another infrastructure-related aspect of service design is any dependencies the service may have on service agents—event-driven intermediary programs capable of transparently intercepting and processing messages sent to or from a service (Figure 4.14). Service agents can be custom-developed or may be provided by the underlying runtime environment, and they form the basis of a pattern appropriately titled Service Agent (543).

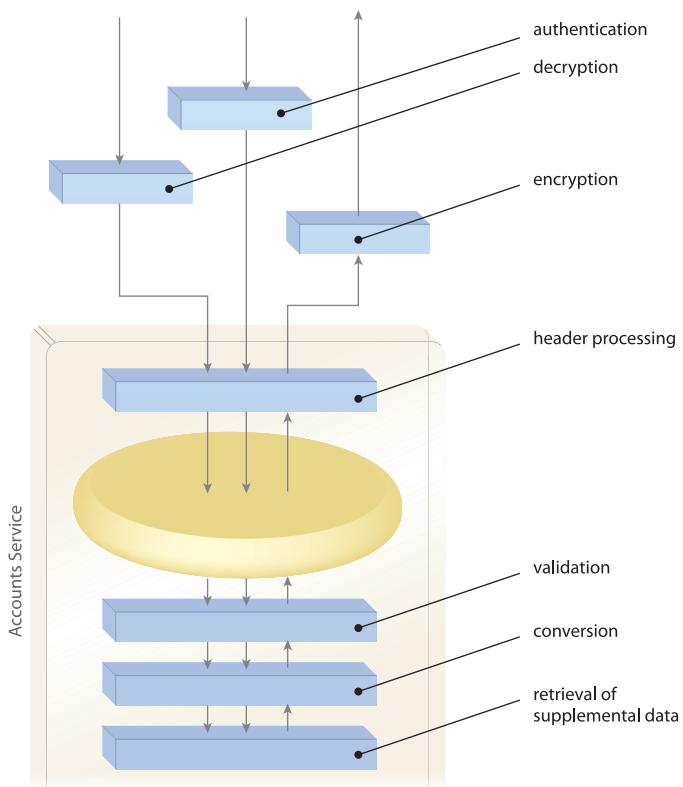


Figure 4.14

A variety of service agents are part of the Accounts service architecture. Some perform general processing of all data while others are specific to input or output data flow.

Within a service architecture the specific agent programs may be identified along with runtime information as to how message contents are processed or even altered by agent involvement. Service agents may themselves have their own architecture specifications that can be referenced by the service architecture.

Service Capabilities

A key consideration with any service architecture is the fact that the functionality offered by a service resides within one or more individual capabilities (Figure 4.15). This often requires the architecture definition itself to be taken to the capability level.

Each service capability encapsulates its own piece of logic, although the underlying logic may itself be modularized allowing different capabilities to share the same routines. Some of this logic may be custom-developed for the service, whereas other capabilities may represent or need to access one or more backend resources (including other services). Therefore, individual capabilities end up with their own, individual designs that may need to be so detailed that they are documented as separate “capability architectures,” all of which relate back to the parent service architecture.

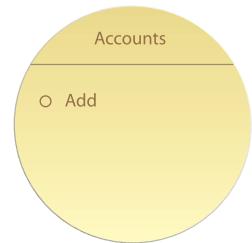


Figure 4.15

The Accounts service with an Add capability.

Service Composition Architecture

The fundamental purpose of delivering a series of independent services is so that they can be combined into service compositions—fully functional solutions capable of automating larger, more complex business tasks (Figure 4.16).

Each service composition has a corresponding *service composition architecture*. In much the same way an application architecture for a distributed system includes the individual architecture definitions of its components, this form of architecture encompasses the service architectures of all participating services (Figure 4.17).

NOTE

Standard composition terminology defines two basic roles that services can assume within a composition. The service responsible for composing others takes on the role of *composition controller*, whereas composed services are referred to as *composition members*.

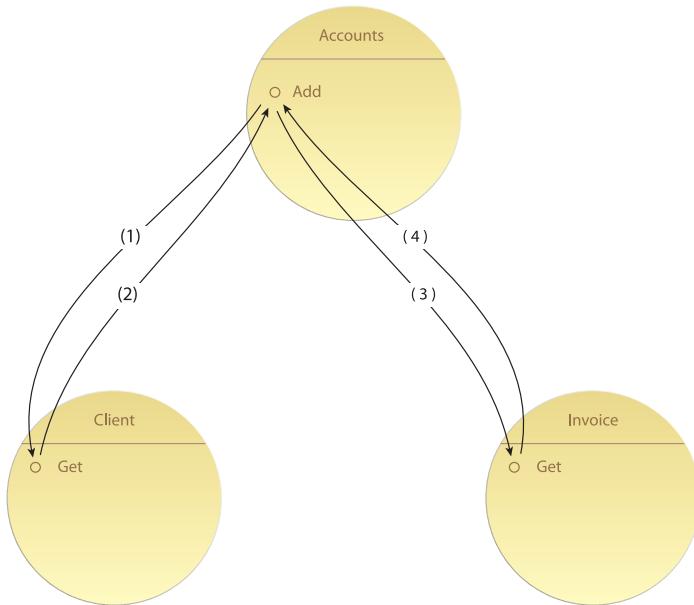
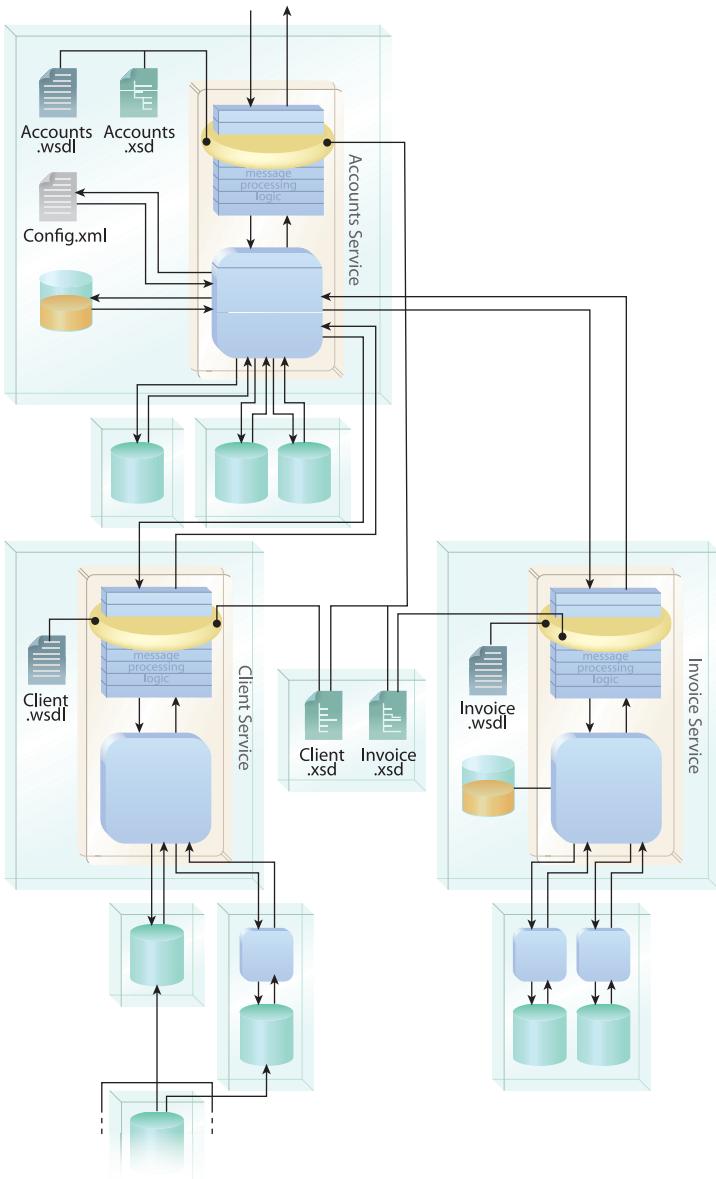


Figure 4.16

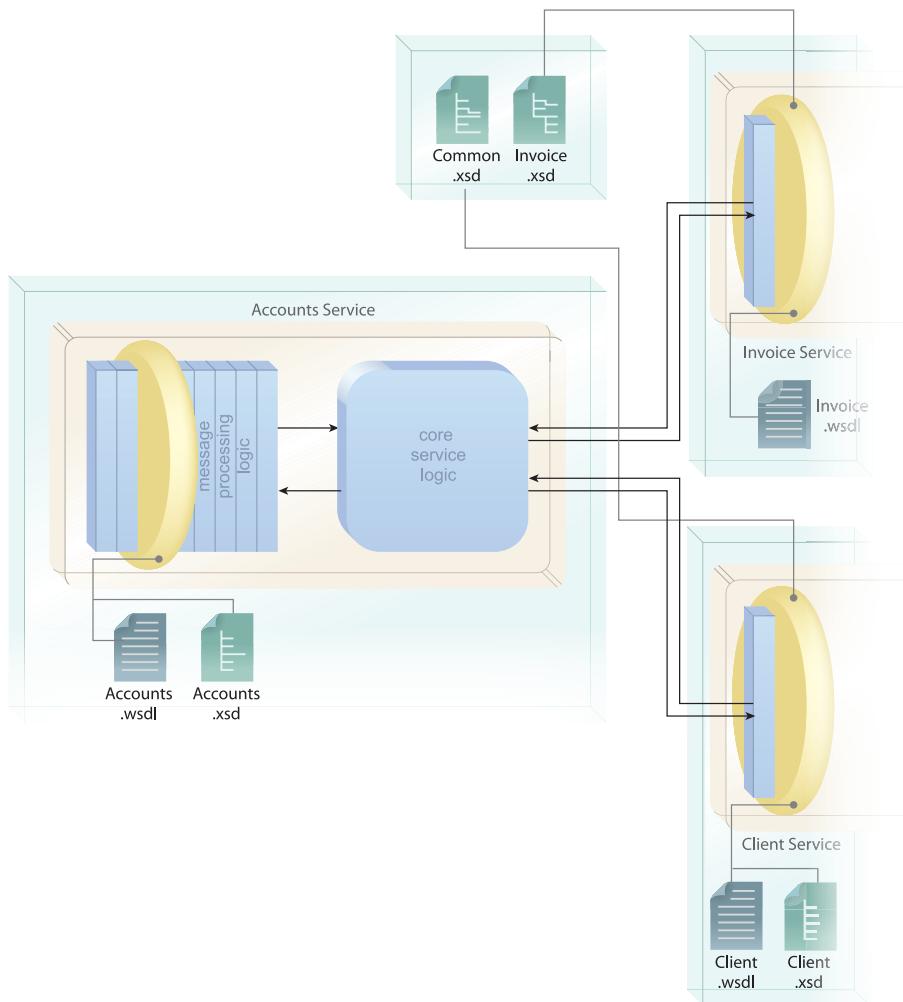
The Accounts service composition from a modeling perspective. The numbered arrows indicate the sequence of data flow and service interaction required for the Add capability to compose capabilities within the Client and Invoice services.

A composition architecture (especially one that composes services that encapsulate disparate legacy systems) may be compared to a traditional integration architecture. This comparison is usually only valid in scope, as the design considerations emphasized by service-orientation ensure that the design of a service composition is much different than that of integrated applications.

For example, one difference in how composition architectures are documented is in the extent of detail they include about reusable services involved in the composition. Because these types of service architecture specifications are often guarded (as per the requirements raised by the Service Abstraction principle), a composition architecture may only be able to make reference to the technical interface and service-level agreement (SLA) published as part of the service's public contract (Figure 4.18).

**Figure 4.17**

The same Accounts service composition from Figure 4.16 viewed from a physical architecture perspective illustrating how each composition member's underlying resources provide the functionality required to automate the process logic represented by the Accounts Add capability.

**Figure 4.18**

The physical service architecture view from Figure 4.17 is not available to the designer of the Accounts service. Instead, only the information published in the contracts for the Invoice and Client services can be accessed and referenced in the Account service composition architecture.

NOTE

Even though compositions are comprised of services, it is actually the service capabilities that are individually invoked and executed in order to carry out the composition logic. This is why design patterns, such as Capability Composition (521) and Capability Recomposition (526) make specific reference to the composed capability (as opposed to the composed service).

Nested Compositions

Another rather unique aspect of service composition architecture is that a composition may find itself a nested part of a larger parent composition, and therefore one composition architecture may encompass or reference another (Figure 4.19).

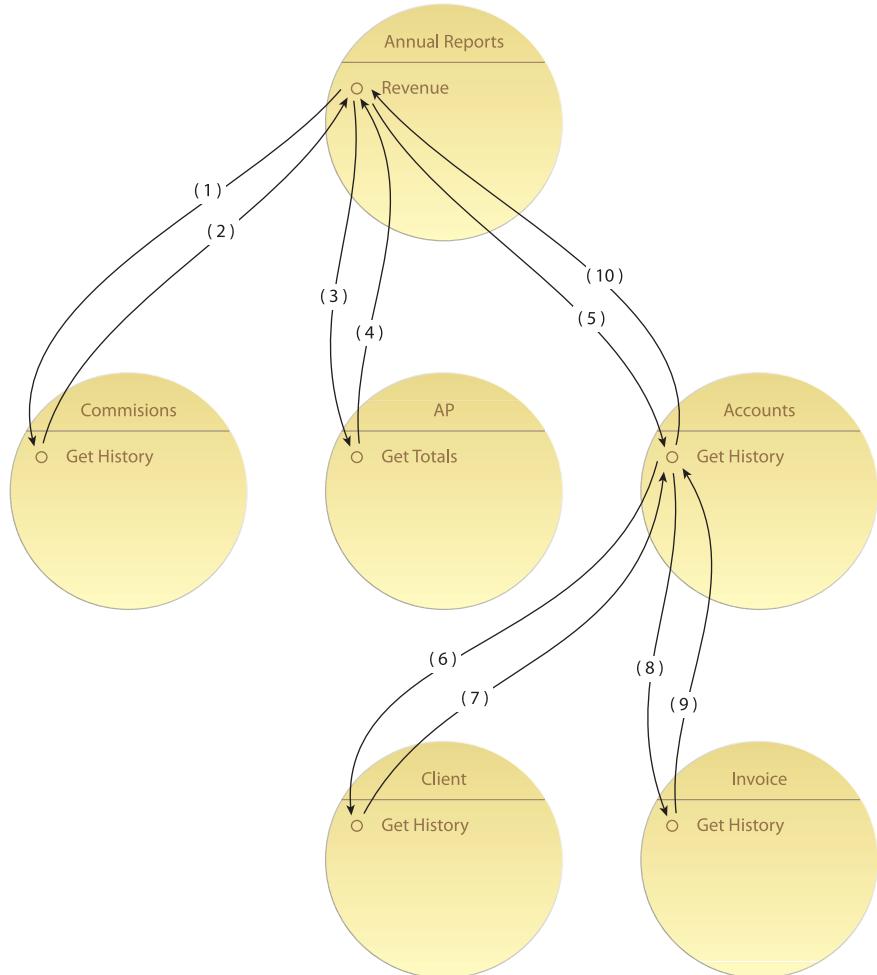


Figure 4.19

The Accounts service finds itself nested within the larger Annual Reports composition that composes the Accounts Get History capability which, in turn, composes capabilities within the Client and Invoice services.

Task Services and Alternative Compositions

Service composition architectures are much more than just an accumulation of individual service architectures (or contracts). A newly created composition is usually accompanied by a task-specific service that is positioned as the composition controller. The details of this service are less private, and its design is an integral part of the architecture because it usually provides most of the composition logic required.

Furthermore, the business process the service is required to automate may involve the need for composition logic capable of dealing with multiple runtime scenarios (exception-related or otherwise), each of which may result in a different composition configuration (Figure 4.20). These scenarios and their related service activities and message paths are a common part of composition designs. They need to be understood and mapped out in advance so that the composition logic encapsulated by the task service is fully prepared to deal with the range of runtime situations it faces.

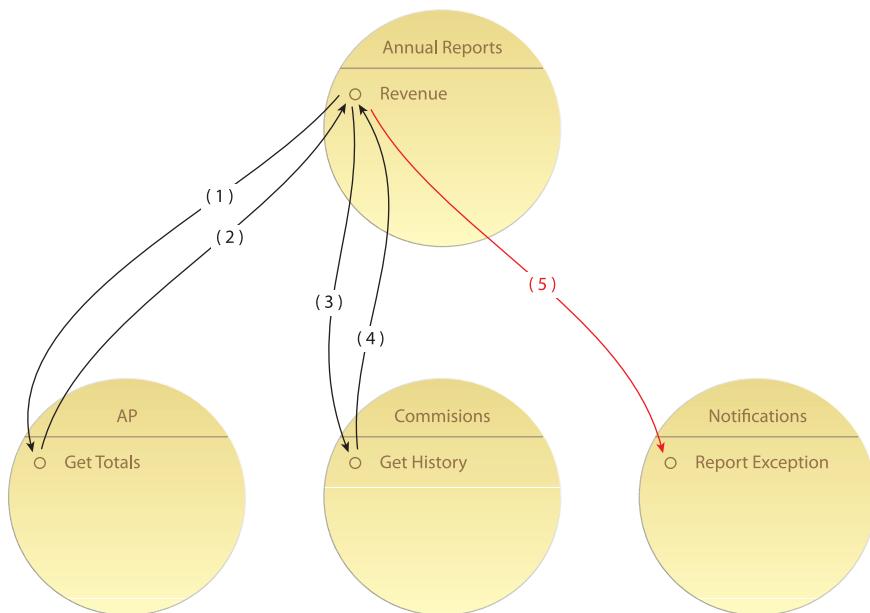


Figure 4.20

A given business process may need to be automated by a range of service compositions in order to accommodate different runtime scenarios. In this case, alternative composition logic within the Annual Report's Revenue capability kicks in to deal with an exception condition. As a result, the Notification service is invoked prior to the Accounts service even being included in the composition. (This scenario represents an alternative composition design to the one shown in Figure 4.19.)

Compositions and Infrastructure

A composition architecture will be heavily dependent on the activity management features of the underlying runtime environment responsible for hosting the services participating in the composition. Security, transaction management, reliable messaging, and other infrastructure extensions, such as support for sophisticated message routing, may all find their way into a typical composition architecture specification. Numerous patterns in this book address these types of extensions.

NOTE

It's often difficult to determine where a service architecture ends and where a composition architecture begins. One service can compose others, which can make a given composition architecture seem like an extension of a service architecture. Often these lines are drawn when the Service Abstraction principle is applied, thereby defining clean boundaries around service architectures that are hidden from the overall composition design.

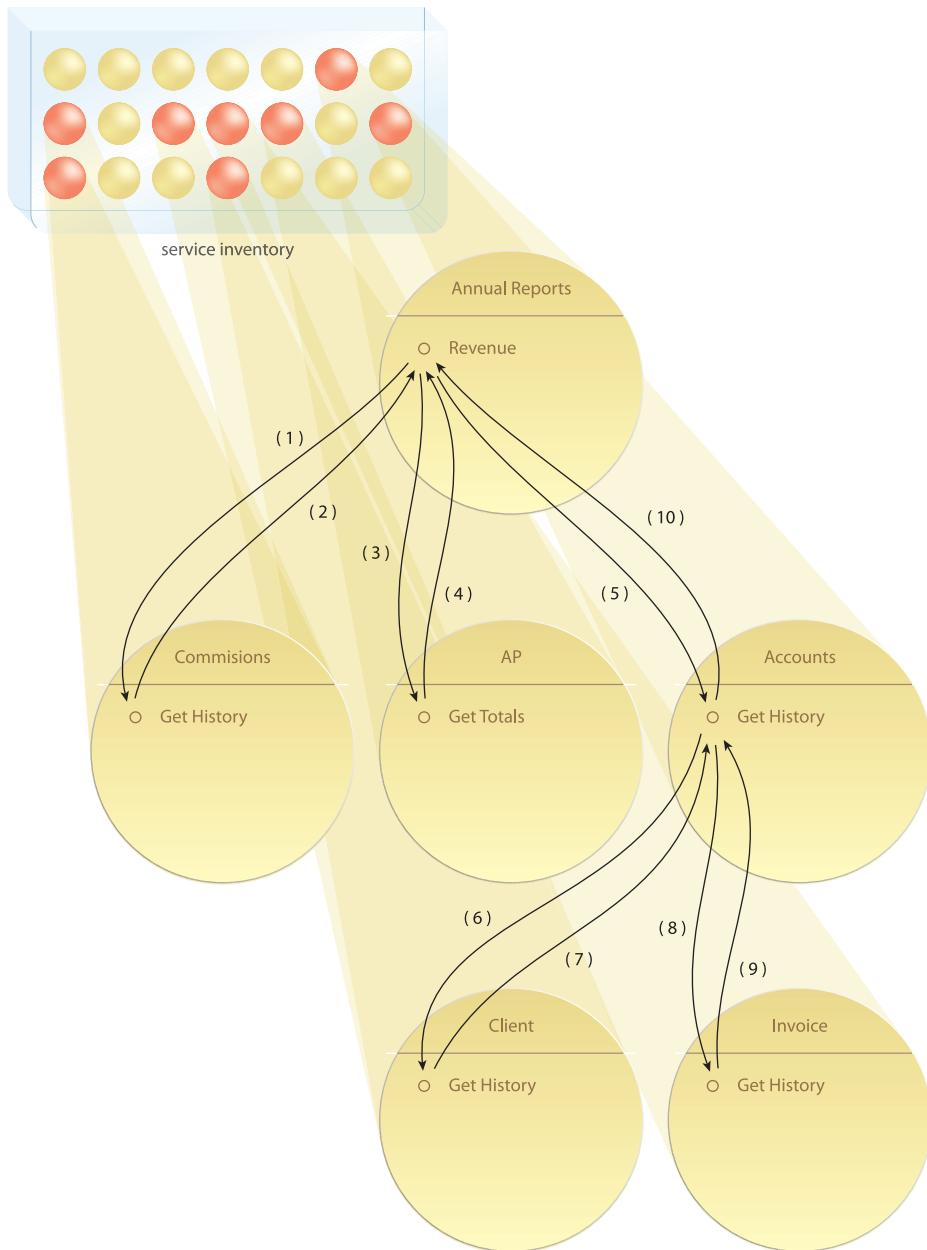
Service Inventory Architecture

Services delivered independently or as part of compositions by different IT projects risk introducing redundancy and non-standardization. This can lead to a non-federated enterprise in which clusters of services mimic an environment comprised of traditional siloed applications.

The result is that though often classified as a service-oriented architecture, many of the traditional challenges associated with design disparity, transformation, and integration continue to emerge and undermine strategic service-oriented computing goals.

As first explained in Chapter 3, a service inventory is a collection of independently standardized and governed services delivered within a pre-defined architectural boundary. This collection represents a meaningful scope that exceeds the processing boundary of a single business process and ideally spans numerous business processes. The scope and boundary of a service inventory architecture can vary, as explained in the Enterprise Inventory (116) and Domain Inventory (123) pattern descriptions.

Ideally, the service inventory is first conceptually modeled, leading to the creation of a service inventory blueprint. It is often this blueprint that ends up defining the required scope of the architecture type referred to as a service inventory architecture (Figure 4.21).

**Figure 4.21**

Ultimately, the services within an inventory can be composed and recomposed, as represented by different composition architectures. To that end, many of the design patterns in this book need to be consistently applied within the boundary of the service inventory.

From an enterprise design perspective, the service inventory can represent a concrete boundary for a standardized architecture implementation. This means that because the services within an inventory are standardized, so are the technologies and extensions provided by the underlying architecture.

As evidenced by the inventory boundary design patterns in Chapter 6, the scope of a service inventory can be enterprise-wide, or it can represent a domain within the enterprise. For that reason, this type of architecture is not called a “domain architecture.” It relates to the scope of the inventory boundary, which may encompass multiple domains.

NOTE

When the term “SOA” or “SOA implementation” is used, it is most commonly associated with the scope of a service inventory. In fact, with the exception of some design patterns that address cross-inventory exchanges, most of the patterns in this book are expected to be applied within the boundary of a service inventory.

It is difficult to compare a service inventory architecture with traditional types of architecture because the concept of an inventory has not been common. The closest candidate would be an integration architecture that represents some significant segment of an enterprise. However, this comparison would be only relevant in scope, as service-orientation design characteristics and related standardization efforts strive to turn a service inventory into a highly homogenous environment.

NOTE

For more information about how service-orientation differs from and attempts to address the primary challenges associated with silo-based, standalone application environments, see [SOAPrinciples.com](#). To learn more about defining service inventory blueprints, visit [SOAMethodology.com](#).

Service-Oriented Enterprise Architecture

This form of technology architecture essentially represents all service, service composition, and service inventory architectures that reside within a specific IT enterprise.

A service-oriented enterprise architecture is comparable to a traditional enterprise technical architecture only when most or all of an enterprise's technical environments are service-oriented. Otherwise it may simply be a documentation of the parts of the enterprise that have adopted SOA, in which case it exists as a subset of the parent enterprise technology architecture.

In multi-inventory environments or in environments where standardization efforts were not fully successful, a service-oriented enterprise architecture specification will further document any transformation points and design disparity that may also exist.

Additionally, the service-oriented enterprise architecture can further establish enterprise-wide design standards and conventions to which all service, composition, and inventory architecture implementations need to comply, and which may also need to be referenced in the corresponding architecture specifications. (Canonical Resources (237) represents a pattern that may introduce such standards.)

NOTE

This chapter is focused on technology architecture only. It is worth pointing out that a “complete” service-oriented enterprise architecture would encompass both the technology and business architecture of an enterprise (much like traditional enterprise architecture).

Architecture Types and Scope

Figure 4.22 illustrates how each of the previously described service-oriented architecture types establishes its own scope. Service architectures fall within composition architectures and both are natural parts of a service inventory architecture. The service-oriented enterprise architecture represents a parent architecture that encompasses all others.

Architecture Types and Inheritance

The environment and conventions established by the enterprise are carried over into individual service inventory architecture implementations that may reside within a

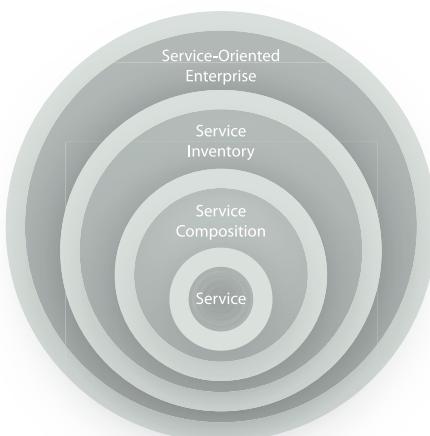


Figure 4.22

A layered model of how service-oriented architecture types encompass each other. This view highlights the common levels of SOA that exist within a typical enterprise.

single enterprise environment. These inventories further introduce new and more specific architectural elements (such as runtime platforms and middleware) that then form the foundation of service and composition architectures implemented within the inventory boundary.

As a result, a natural form of architectural inheritance is formed. This relationship between architecture types is good to keep in mind as it can identify potential (positive and negative) dependencies that may exist.

Other Forms of Service-Oriented Architecture

All of the architecture types explored so far relate mostly to a private IT enterprise environment. While these represent the most common variations, they are by no means the only ones. The following two sections discuss some examples of additional architecture types with different scopes and characteristics.

Inter-Business Service Architecture

This is an architecture that spans enterprises and therefore is prone to encompass diverse environments and incompatible design conventions. A focal point is often the use of transformation technologies, security, and access to subsets of inventory logic.

Note that an inter-business service architecture is different from a traditional B2B environment. Data exchange is designed around communication across partner service inventories via predefined service endpoints and can include specialized compositions that also span inventories.

Service-Oriented Community Architecture

With the advent of community-centric data exchange standards, such as the Web Services Choreography Description Language (WS-CDL) and a growing marketplace of third-party services, the option is there to define a service-oriented architecture dedicated to collaboration among community members.

NOTE

Patterns, such as Service Perimeter Guard (394), Inventory Endpoint (260), and many of the security patterns in Chapters 13 and 20, are often used to support this SOA type.

Additional variations of service-oriented architecture can exist. For example, hybrid architectures comprised of a combination of traditional and service-oriented elements may be created as a result of pilot or partially completed transition projects, or they may exist as an intermediate architecture while a larger-scale migration is still underway.

SUMMARY OF KEY POINTS

- A service architecture represents the technology architecture that pertains to a service and any of the enterprise resources its capabilities may need to access or rely upon.
 - A service composition architecture corresponds to a service composition and therefore encompasses the individual architectures of its member services as well as any additional architectural elements that may be required to carry out the composition logic.
 - A service inventory architecture typically represents a significant scope that encompasses all service and composition architectures related to the boundary of a pre-defined service inventory.
 - A service-oriented enterprise architecture represents all service inventories within an enterprise and further defines cross-inventory communication options and enterprise design standards.
-

4.4 The End Result of Service-Orientation

Automated business communities and the IT industry have an endless bi-directional relationship where each influences the other. Business demands and trends create automation requirements that the IT community strives to fulfill. New method and technology innovations produced by the IT community help inspire organizations to improve their existing business and even try out new lines of business. (The advent of the Internet is a good example of the latter.)

The IT industry has been through the cycle depicted in Figure 4.23 many times. Each iteration has brought about change and generally an increase in the sophistication and complexity of technology platforms.

Sometimes a series of iterations through this progress cycle leads to a foundational shift in the overall approach to automation and computing itself. The emergence of major platforms and frameworks, such as object-orientation and enterprise application integration,

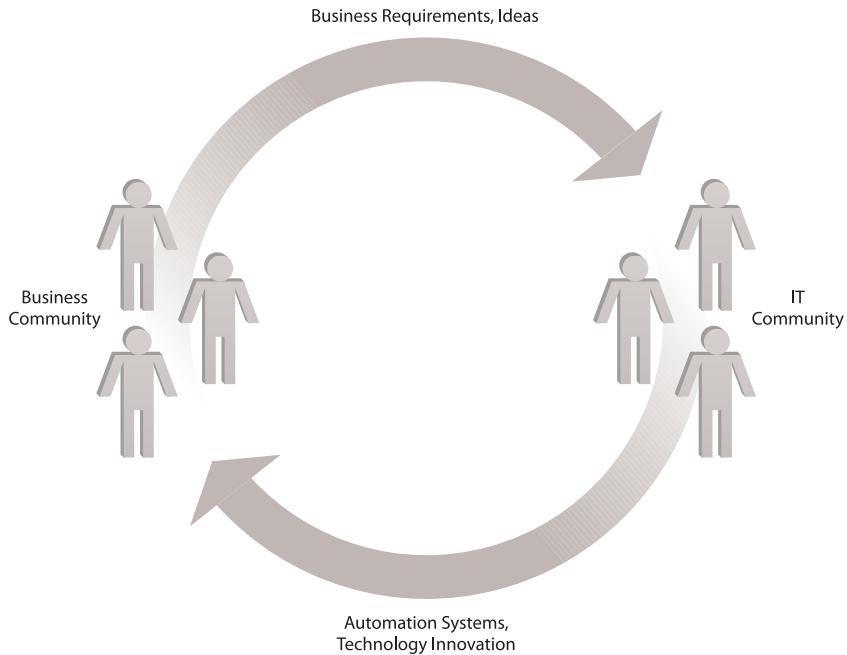


Figure 4.23

The endless progress cycle establishes the dynamics between the business and IT communities.

are examples of this. Significant changes like these represent an accumulation of technologies and methods and can therefore be considered landmarks in the evolution of IT itself. Each also results in the formation of distinct technology architecture requirements.

Service-oriented computing is no exception. The platform it establishes provides the potential to achieve significant strategic benefits that are a reflection of what business communities are currently demanding, as represented by the following previously described goals:

- Increased Intrinsic Interoperability
- Increased Federation
- Increased Vendor Diversification Options
- Increased Business and Technology Domain Alignment
- Increased ROI
- Increased Organizational Agility
- Reduced IT Burden

It is the target state resulting from the attainment of these strategic goals that an adoption of service-orientation attempts to achieve. In other words, these goals represent the desired end-result of applying the method of service-orientation.

How then does this relate to service-oriented technology architecture? Figure 4.24 hints at how the pursuit of these specific goals results in a series of impacts onto all architecture types brought upon by the application of service-orientation.

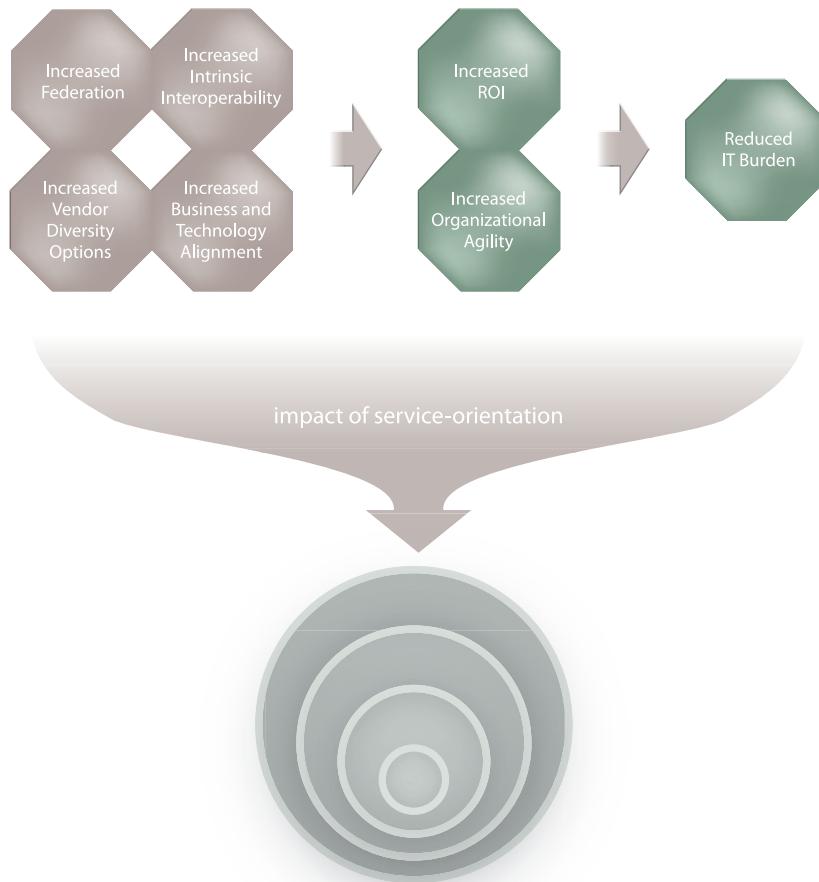


Figure 4.24

The common strategic goals and benefits of service-oriented computing are realized through the application of service-orientation. This, in turn, impacts the demands and requirements placed upon the four types of service-oriented technology architectures. (Note that the three goals on the right [green] represent the ultimate target benefits achieved when attaining the four goals on the left [gray].)

NOTE

For those of you interested in how each of the strategic goals relates to specific design patterns, see Chapter 23.

Almost all of the design patterns in this book are specifically intended to support the application of service-orientation by solving common problems that may arise as a result of the impact placed upon the different architecture types. This is an important perspective to keep in mind when working with SOA design patterns, as it is always helpful to understand that all patterns in this catalog share this common objective.

Ultimately, the successful implementation of service-oriented architectures will support and maintain the benefits associated with the strategic goals of service-oriented computing. As concluded by Figure 4.25, the progress cycle that continually transpires between business and IT communities results in constant change. Standardized, optimized, and overall robust service-oriented architectures fully support and even enable the accommodation of this change as a natural characteristic of a service-oriented enterprise.

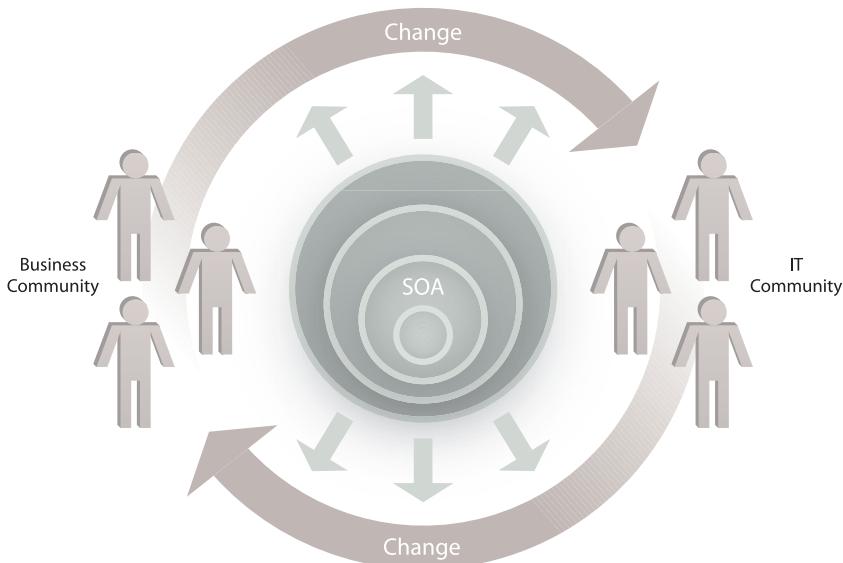


Figure 4.25

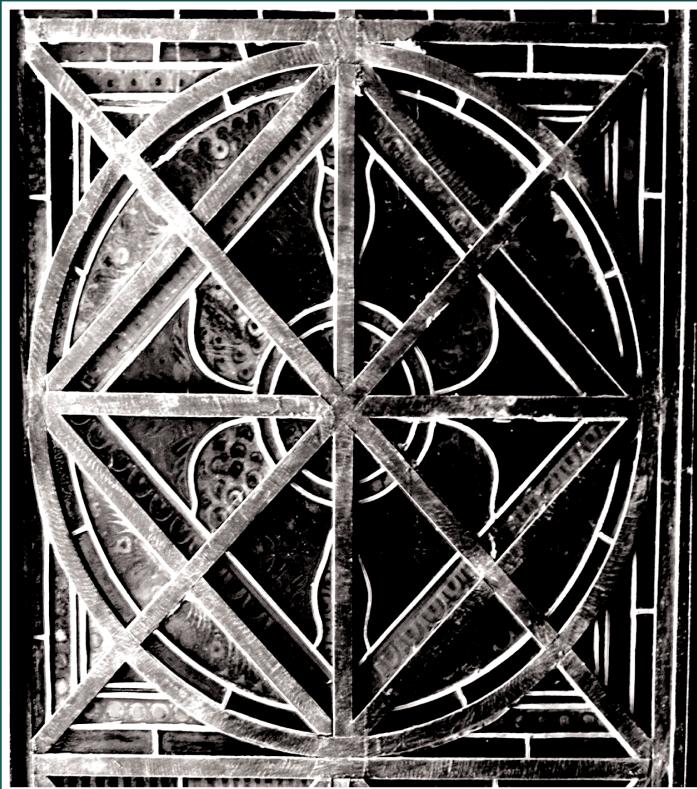
Ultimately, service-orientation and service-oriented technology architectures support the two-way dynamic between business and IT communities, allowing each to introduce or accommodate change throughout an endless cycle.

SUMMARY OF KEY POINTS

- Service-orientation places various demands on all types of service-oriented architectures.
 - Specific requirements can be defined when studying each of the goals of service-oriented computing.
-

This page intentionally left blank

Chapter 5



Understanding SOA Design Patterns

5.1 Fundamental Terminology

5.2 Historical Influences

5.3 Pattern Notation

5.4 Pattern Profiles

5.5 Patterns with Common Characteristics

5.6 Key Design Considerations

The first step to forming an effective working relationship with SOA design patterns is attaining a sound comfort level with pattern-related terminology and notation. This provides us with the knowledge required to navigate through the upcoming chapters with insight as to how the patterns can be applied individually and in various combinations.

Purpose of this Introductory Chapter

This important chapter covers these fundamental topics and further describes how design pattern descriptions are organized into standardized profiles. The remaining sections single out specific pattern types and discuss some common design considerations.

5.1 Fundamental Terminology

What's a Design Pattern?

The simplest way to describe a pattern is that it provides a proven solution to a common problem individually documented in a consistent format and usually as part of a larger collection.

The notion of a pattern is already a fundamental part of everyday life. Without acknowledging it each time, we naturally use proven solutions to solve common problems each day. Patterns in the IT world that revolve around the design of automated systems are referred to as *design patterns*.

Design patterns are helpful because they:

- represent field-tested solutions to common design problems
- organize design intelligence into a standardized and easily “referencable” format
- are generally repeatable by most IT professionals involved with design
- can be used to ensure consistency in how systems are designed and built
- can become the basis for design standards
- are usually flexible and optional (and openly document the impacts of their application and even suggest alternative approaches)

- can be used as educational aids by documenting specific aspects of system design (regardless of whether they are applied)
- can sometimes be applied prior and subsequent to the implementation of a system
- can be supported via the application of other design patterns that are part of the same collection

Furthermore, because the solutions provided by design patterns are proven, their consistent application tends to naturally improve the quality of system designs.

Let's provide a simple (non SOA-related) example of a design pattern that addresses a user interface design problem:

Problem: *How can users be limited to entering one value of a set of predefined values into a form field?*

Solution: *Use a drop-down list populated with the predefined values as the input field.*

What this example also highlights is the fact that the solution provided by a given pattern may not necessarily represent the only suitable solution for that problem. In fact, there can be multiple patterns that provide alternative solutions for the same problem. Each solution will have its own requirements and consequences, and it is up to the practitioner to choose.

In the previous example, a different solution to the stated problem would be to use a list-box instead of a drop-down list. This alternative would form the basis of a separate design pattern description. The user-interface designer can study and compare both patterns to learn about the benefits and trade-offs of each. A drop-down list, for instance, takes up less space than a list box but requires that a user always perform a separate action to access the list. Because a list box can display more field lines at the same time, the user may have an easier time locating the desired value.

NOTE

Even though design patterns provide proven design solutions, their mere use cannot guarantee that design problems are always solved as required. Many factors weigh in to the ultimate success of using a design pattern, including constraints imposed by the implementation environment, competency of the practitioners, diverging business requirements, and so on. All of these represent aspects that affect the extent to which a pattern can be successfully applied.

What's a Compound Pattern?

A compound pattern is a coarse-grained pattern comprised of a set of finer-grained patterns. Compound patterns are explained in detail at the beginning of Chapter 22.

What's a Design Pattern Language?

A *pattern language* is a set of related patterns that act as building blocks in that they can be carried out in *pattern sequences* (or *pattern application sequences*), where each subsequent pattern builds upon the former. As explained shortly in the *Historical Influences* section, the notion of a pattern language originated in building architecture as did the term “pattern sequence” used in association with the order in which patterns can be carried out.

Some pattern languages are open-ended, allowing patterns to be combined into a variety of creative sequences, while others are more structured whereby groups of patterns are presented in a suggested application sequence. In this case, the pattern sequence is generally based on the granularity of the patterns, in that coarser grained patterns are applied prior to finer-grained ones that then build upon or extend the foundation established by the coarse-grained patterns. In these types of pattern languages, the manner in which patterns can be organized into pattern sequences may be limited to how they are applied within their groups.

Structured pattern languages are helpful because they:

- can organize groups of field-tested design patterns into proposed, field-tested application sequences
- ensure consistency in how particular design goals are achieved (because by carrying out sets of inter-dependent patterns in a proven order, the quality of the results can be more easily guaranteed)
- are effective learning tools that can provide insight into how and why a particular method or technique should be applied as well as the effects of its application
- provide an extra level of depth in relation to pattern application (because they document the individual patterns plus the cumulative effects of their application)
- are flexible in that the ultimate pattern application sequence is up to the practitioner (and also because the application of any pattern within the overall language can be optional)

This book in its entirety provides an open-ended, master pattern language for SOA. The extent to which different patterns are related can vary, but overall they share a common

objective and endless pattern sequences can be explored. The relationship diagrams explained in the upcoming *Pattern Relationship Figures* section will often hint at common application sequences for a given pattern.

Chapters 6 and 11 single out sets of closely related patterns and structure them into groups organized into recommended application sequences that essentially establish primitive design processes. As a result, these collections of patterns could be classified as “mini” structured pattern languages that are still part of the overall master pattern language.

What's a Design Pattern Catalog?

A design pattern catalog is simply a documented collection of related design patterns. Therefore, this book is also referred to as a catalog for design patterns associated with SOA and service-orientation.

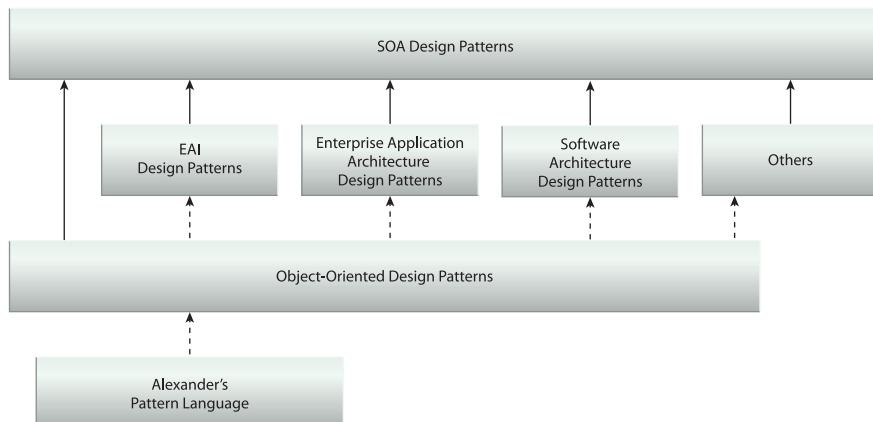
SUMMARY OF KEY POINTS

- A design pattern is a formal documentation of a proven solution to a common problem.
 - A design pattern language is a group of related design patterns that can be applied in a variety of creative application sequences.
 - A design pattern catalog is a collection of related design patterns documented together.
 - This book contains a design pattern catalog that documents a master pattern language for SOA.
-

5.2 Historical Influences

Because service-orientation has deep roots in past distributed computing design platforms, many of the SOA design patterns have origins and influences that can be traced back to established design concepts, approaches, and previously published design pattern catalogs.

As illustrated in Figure 5.1, object-orientation, EAI, enterprise application architecture, and software architecture in general represent areas for which well-recognized design pattern catalogs exist, each of which has influenced design patterns in this book. Starting with the original pattern language created by Christopher Alexander, let's briefly discuss these influences separately.

**Figure 5.1**

The primary influences of SOA design patterns.

Alexander's Pattern Language

It's been well documented how the notion of the design pattern owes its existence to the work of Christopher Alexander. Just about every design pattern publication pays tribute to Alexander's pattern language as a fundamental influence and source of inspiration.

Alexander pioneered the concept of patterns in relation to building architecture and related areas, such as city and community structure. He documented a collection of patterns and organized them into a pre-defined series he called a "sequence." The result was an architectural pattern language that inspired the IT community to create their own patterns for the design of automated systems.

Alexander's work is more than just a historical footnote for design patterns; it provides insight into how patterns in general should and should not be structured and organized.

For example, some lessons learned from Alexander's work include:

- *Pattern language sequences need to add value.* Often related patterns are better documented independently from each other even if there is some potential for them to be organized into a sequence. The primary purpose of any application sequence established by a pattern language is *not* to provide a logical organization for a set of related patterns but to demonstrate a proven process that provides value on its own.
- *Patterns do not need to be normalized.* There is often a perception that each design pattern should own an individual domain. In reality, the problem and solution space represented by individual patterns sometimes naturally overlaps. For example, you can easily have two patterns that propose different solutions to the same problem.

Beyond just the idea of organizing solutions into a pattern format, Alexander helped advocate the importance of clarity in how pattern catalogs need to be documented. He preached that patterns need to be individually clear as to their purpose and applicability and that pattern languages need to further communicate the rationale behind any sequences they may propose.

NOTE

As provided by research from Dr. Peter H. Chang from Lawrence Technological University, earlier origins also exist. For example, George Polya published the book *How to Solve It* (Princeton University Press) back in 1945, which included a “problem solving plan” that can be viewed at www.math.utah.edu/~pa/math/polya.html (based on the second edition released in 1957). Furthermore, Marvin Minsky published the paper *Steps Toward Artificial Intelligence* for MIT in 1960 that included coverage of pattern recognition and made further reference to Polya’s work.

Object-Oriented Patterns

A variety of design patterns in support of object-orientation surfaced over the past 15 years, the most recognized of which is the pattern catalog published in *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, Helm, Johnson, Vlissides; Addison-Wesley, 1995). This set of 23 patterns produced by the “Gang of Four” expanded and helped further establish object-orientation as a design approach for distributed solutions. Some of these patterns have persisted within service-orientation, albeit within an augmented context and new names.

For example, the following patterns in this book are related:

- Capability Composition (521) is associated with Composite
- Service Façade (333) is derived from Façade
- Legacy Wrapper (441) is derived from Adapter
- Non-Agnostic Context (319) is associated with Mediator
- Decoupled Contract (401) is associated with Bridge

Concepts established by several additional object-orientation patterns have factored into other SOA patterns. The incorporation of these patterns within service-orientation is a testament to their importance and evidence of how object-orientation as a whole has influenced the evolution of SOA.

er relevant object-oriented-related influence is the paper “Using Pattern Languages for Object-Oriented Programs” published by Kent Beck and Ward Cunningham for the OOPSLA conference. This paper is notable not only for its brevity, but for its vision of explicit emphasis on the use of sequences in organizing patterns.

NOTE

The comparative analysis in Chapter 14 of *SOA Principles of Service Design* provides a study of how object-oriented design concepts and principles relate to service-orientation.

are Architecture Patterns

Design patterns became a mainstream part of IT, a set of important books emerged establishing formal conventions for pattern documentation and providing a series of comprehensive design patterns for software architecture in general. These pattern catalogs were developed in five separate volumes over a period of a dozen years as part of the *Patterns of Enterprise Application Architecture* series (F. Buschmann, K. Henney, M. Kircher, R. Meunier, J. Rohnert, D. Schmidt, P. Sommerlad, M. Stal, Wiley 1996–2007).

Because of the general nature of the patterns, the contributions made by this series are too numerous to document individually. Here are some examples of how SOA design patterns relate:

Service Layers (143) is associated with Layers

Service Broker (707) compound pattern is associated with Broker

Concurrent Contracts (421) is associated with Extension Interface

Metadata Centralization (280) is associated with Lookup

Event-Driven Messaging (599) is derived from Publisher-Subscriber

Process Abstraction (182) is associated with Whole-Part

Economic Service Transaction (623) is associated with Coordinator and Task Coordinator

Partial State Deferral (356) is associated with Partial Acquisition

also worth noting that Volume 4 of the series (entitled *A Pattern Language for Distributed Computing*) focuses on connecting existing patterns relevant to building distributed systems into a larger pattern language. This book documents the roots of various previously published patterns, including those that are part of other pattern catalogs listed in the introduction.

Enterprise Application Architecture Patterns

Distributed computing became an established platform for solution design, an emphasis on enterprise architecture emerged bringing with it its own set of design patterns, many of which built upon object-oriented concepts and patterns. A respected pattern catalog in this field was published in *Patterns of Enterprise Application Architecture* (Fowler, Addison Wesley, 2003).

One might notice that many of the influences originating from enterprise architecture patterns are located in the two pattern languages provided in Chapters 6 and 11. Service Orientation is, at heart, a design paradigm for distributed computing, and although distributed computing still relies and builds upon the fundamental patterns and concepts associated with enterprise application architecture in general.

For example, the following patterns in this book are related:

Service Encapsulation (305) is associated with Gateway and Service Layer

Decoupled Contract (401) is associated with Separated Interface

Service Façade (333) is derived from Remote Façade

Stateful Services (248) is derived from Server Session State

Partial State Deferral (356) is derived from Lazy Load

State Repository (242) is derived from Database Session State

Combining these types of influences can lead to further revelations as to how SOA has evolved into a unique architectural model.

Patterns

Several pattern catalogs centered around the use of messaging to fulfill integration requirements emerged during the EAI era. These patterns establish sound approaches for robust messaging-based communication and address various integration-related challenges.

A recognized publication in this field is *Enterprise Integration Patterns* (Hohpe, Woolf, Addison-Wesley, 2004).

Because EAI is one of the primary influences of service-orientation, this book contains service interaction patterns based on the use of messaging primarily in support of service composition scenarios.

Some examples of SOA patterns related to design patterns documented in *Enterprise Integration Patterns*:

- Service Messaging (533) is derived from Message, Messaging, and Document Message
- Data Model Transformation (671) is derived from Message Translator
- Canonical Schema (158) is associated with Canonical Data Model
- Service Agent (543) is associated with Event-Driven Consumer
- Process Centralization (193) is associated with Process Manager
- Intermediate Routing (549) is derived from Content-Based Router

Several references to additional EAI patterns are interspersed in the upcoming chapters (Chapter 18, in particular).

SOA Patterns

The intention behind this collection of SOA patterns is not to replace or compete with the catalogs provided by previous publications, but instead to build upon and complement them with a catalog focused solely on attaining the strategic goals associated with service-orientated computing.

This catalog is comprised of new patterns, existing patterns that have been augmented, and patterns that are intentionally similar to patterns in other catalogs. The latter group is included so that these patterns can be explained within the context of SOA and to also formally highlight them as a supporting part of the service-orientation design paradigm.

Learning about the design solutions and techniques provided by SOA design patterns can provide insight into the mechanics required to enable service-orientation and also help clarify exactly how SOA represents a distinct architectural model. When exploring these distinctions, it is important to take into account:

- which of the past design techniques are preserved and emphasized
- which of the past design techniques are de-emphasized
- new design techniques
- new approaches to carrying out existing techniques

Note that there are several more useful design patterns in the previously mentioned books which are not mentioned in this pattern catalog. Some provide detailed solutions that are not necessarily specific to SOA, but still very helpful.

SUMMARY OF KEY POINTS

- The pattern language invented by Christopher Alexander inspired the use of design patterns in the IT world.
 - The object-orientation platform has an established set of design patterns that are at the root of several of the patterns in this catalog. Additional influences can be traced back to patterns created for enterprise application architecture, EAI, and general software architecture pattern catalogs.
-

5.3 Pattern Notation

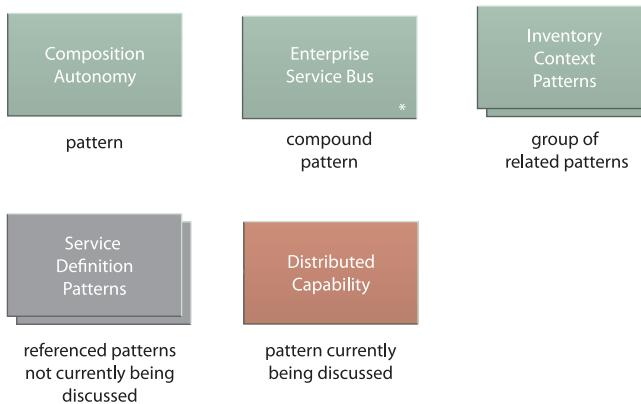
Throughout this book design patterns need to be referenced and explained in text and illustrations. A simple notation is used to consistently represent different types of patterns.

Pattern Symbols

As shown in Figure 5.2, specific symbols are used to represent:

- a design pattern
- a compound design pattern
- a group of related design patterns

Additionally, colors are incorporated to indicate if a displayed design pattern is just being referenced and not actually discussed, versus one that is the current topic of discussion.

**Figure 5.2**

The standard symbols used to represent different types of design patterns and how design patterns relate to the current subject being covered.

Pattern Figures

The symbols displayed in Figure 5.2 are used in the following three primary types of diagrams:

- pattern application sequence figures
- pattern relationship figures
- compound pattern hierarchy figures

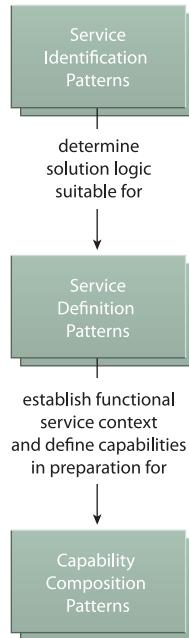
Let's take a closer look at each:

Pattern Application Sequence Figures

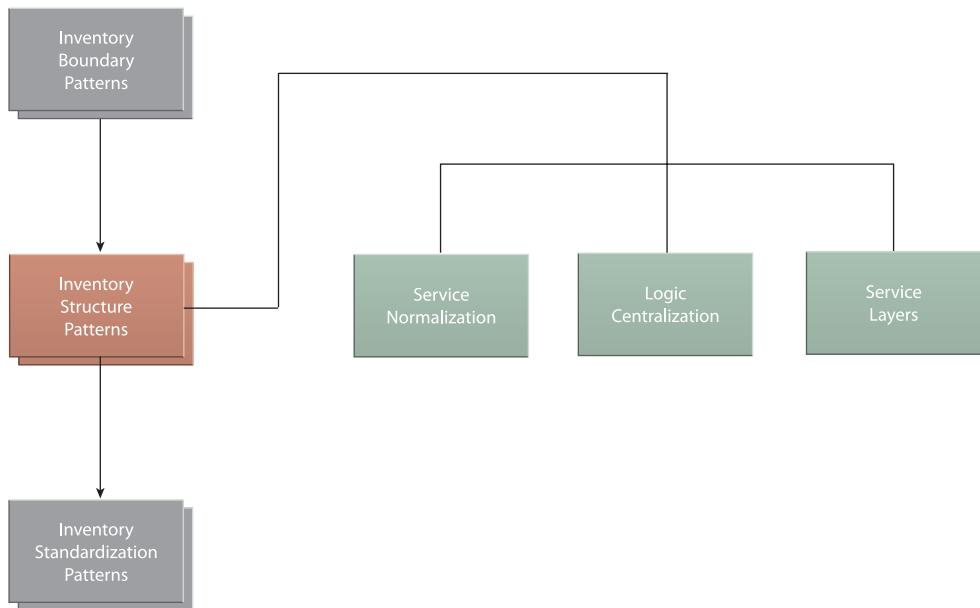
When documenting design pattern languages, it is helpful to display the suggested sequence in which patterns should be applied. Figures 5.3 and 5.4 show pattern application sequences for groups of related patterns and for individual patterns belonging to a particular group, respectively.

Pattern Relationship Figures

As explained in the upcoming *Pattern Profiles* section, this book explores numerous inter-pattern relationships and provides one pattern relationship diagram (Figure 5.5) for each documented design pattern.

**Figure 5.3**

The pattern groups from Chapters 11 and 17 displayed in a recommended application sequence.

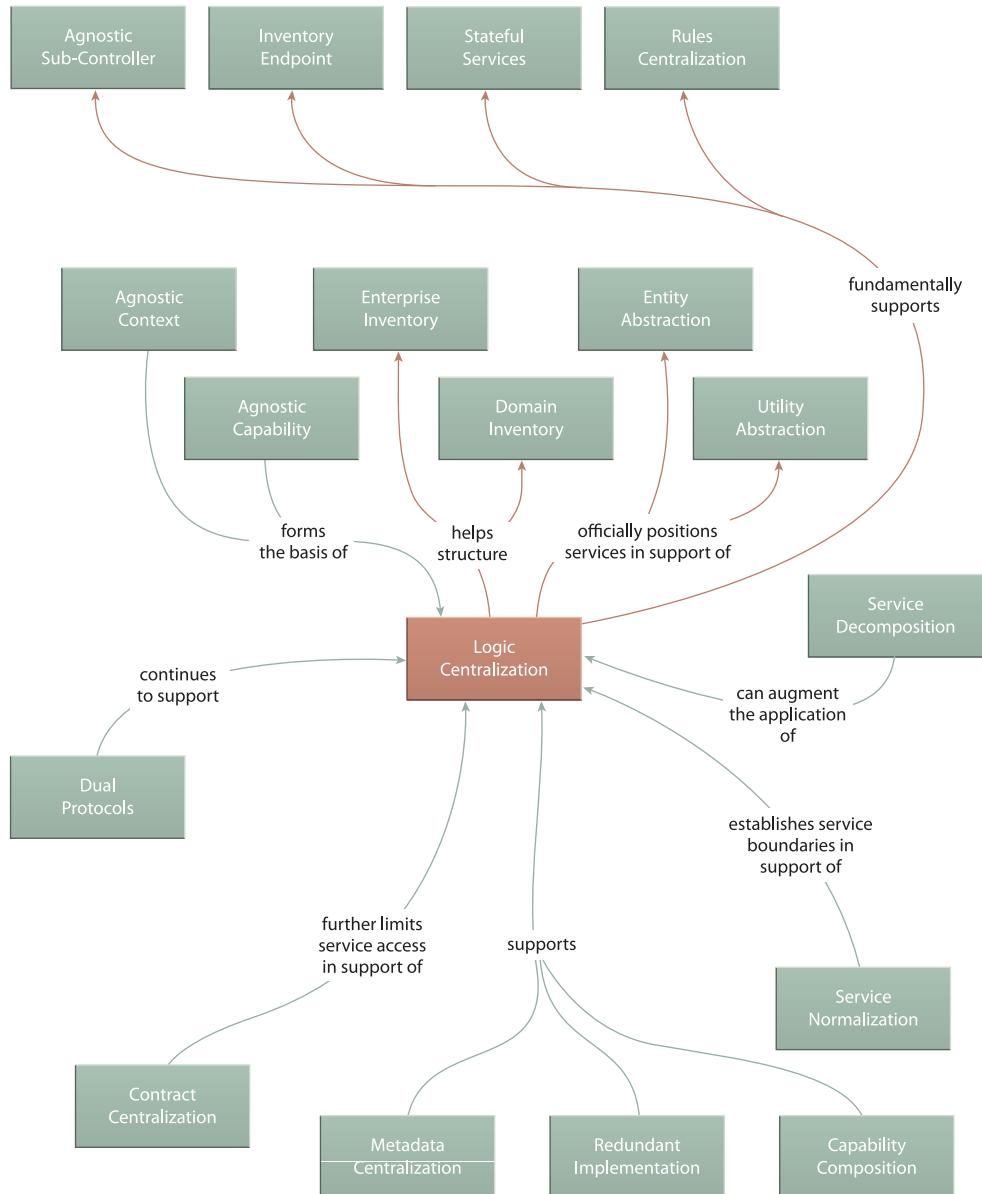
**Figure 5.4**

The inventory structure patterns group from Chapter 6 is highlighted in this diagram. In this case, there is no recommendation as to the order in which the three patterns on the right should be applied.

A style convention applied to all pattern relationship diagrams is the use of color, as follows:

- Each pattern relationship diagram explores the relationships of one pattern. Therefore, that design pattern is highlighted in red, as per the previously established symbol notation.
- Pattern relationships are documented in a unidirectional manner. For relationships where the pattern currently being discussed affects or relates to other patterns, a red line is used along with an arrow pointing to the other pattern. When the relationship line documents how other patterns relate to the current pattern, the lines are green, and the arrows are reversed.

Note that directionality of relationships is preserved in different diagrams. For example, the green relationship line emitting from Service Normalization (131) and pointing to Logic Centralization (136) in the preceding figure would be reversed (and colored red) in the pattern relationship figure for Service Normalization (131).

**Figure 5.5**

An example of a pattern relationship diagram.

Compound Pattern Hierarchy Figures

Compound patterns are comprised of combinations of design patterns. When illustrating a compound pattern, a hierarchical representation is usually required, where the compound pattern name is displayed at the top, and the patterns that comprise the compound are shown underneath.

These types of diagrams (Figures 5.6 and 5.7) can be considered simplified relationship figures in that they only identify which patterns belong to which compound, without getting into the details of how these patterns relate. Compound patterns are documented separately in Chapter 22, but compound hierarchy figures are displayed throughout the upcoming chapters.

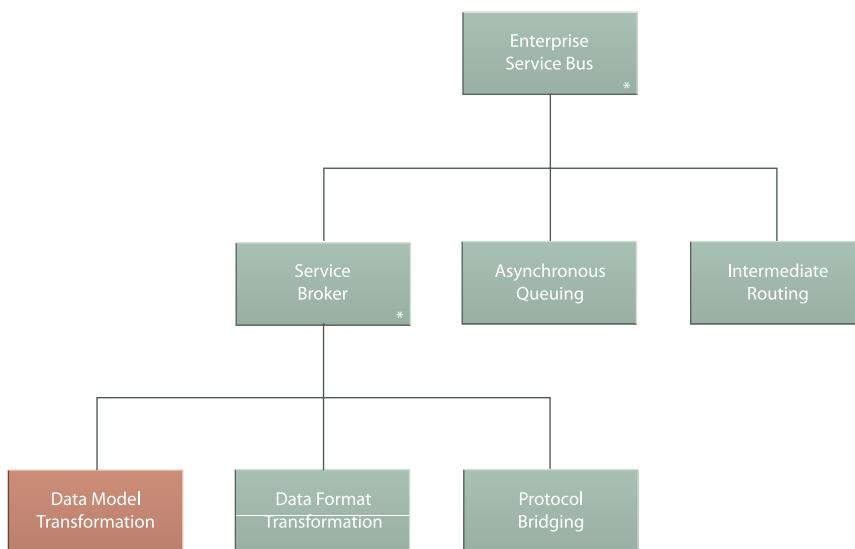


Figure 5.6

Enterprise Service Bus (704) is a compound pattern comprised of several core patterns, one of which is a compound pattern in its own right and therefore represents a nested pattern hierarchy. In this case, Data Model Transformation (671) is highlighted, indicating that it is the current pattern being discussed.

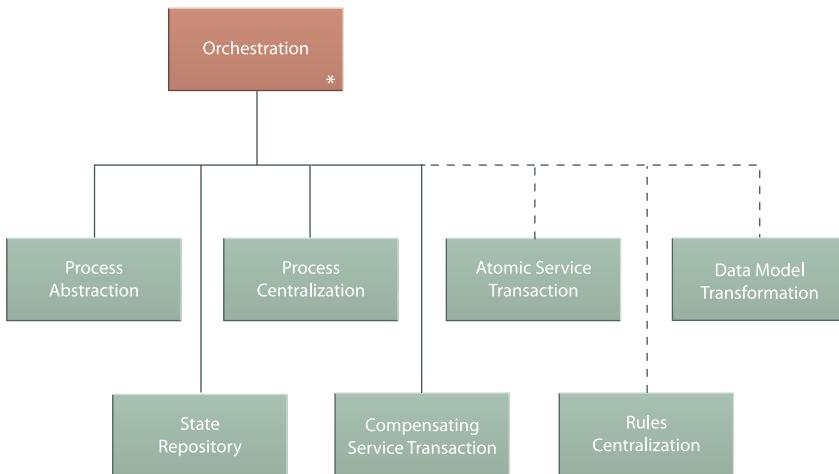


Figure 5.7

There are additional patterns associated with Orchestration (701) that can be considered optional extensions. In this case, the hierarchy lines are dashed.

NOTE

Another notation used for some forms of compound patterns involves showing patterns combined with a plus (“+”) symbol. These diagrams are limited to Chapter 22 and are formally described there.

Capitalization

All design pattern names (including names of compound patterns) are capitalized throughout this book. The names for groups of related patterns are capitalized when displayed in Figures but not when referenced in body text.

Page Number References

As you may have already noticed in earlier parts of this chapter, each pattern name is followed by a page number in parentheses. This number, which points to the first page of the corresponding pattern profile, is provided for quick reference purposes. Its use has become a common convention among pattern catalogs. The only time the number is not displayed is when a pattern name is referenced within that pattern’s profile section.

5.4 Pattern Profiles

Each of the patterns in this catalog is described using the same profile format and structure based on the following parts:

- Requirement
- Icon
- Summary
- Problem
- Solution
- Application
- Impacts
- Relationships
- Case Study Example

The following sections describe each part individually.

Requirement

This is a concise, single-sentence statement that presents the fundamental requirement addressed by the pattern in the form of a question. Every pattern description begins with this statement.

For example:

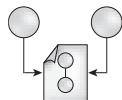
How can a service be designed to minimize the chances of capability logic deconstruction?

Note that the inside cover of this book lists all of the patterns together with their respective requirement statements.

Icon

Each pattern description is accompanied by an icon image that acts as a visual identifier.

An example of a pattern icon:



The icons are displayed together with the requirement statements in each pattern profile as well as on the inside book cover.

Summary

Following the requirement statement, a summary table is displayed, comprised of statements that collectively provide a concise synopsis of the pattern for quick reference purposes.

The following parts of the profile are summarized in this table:

- Problem
- Solution
- Application
- Impacts

Additionally, the profile table provides references to related service-orientation design principles and service-oriented architectural types via the following sections:

- Principles
- Architecture

The parts of the pattern description not represented in the summary table are the *Relationships* and *Case Study Example* sections.

NOTE

All pattern summary tables in this book are also published online at SOAPatterns.org.

Problem

The issue causing a problem and the effects of the problem are described in this section, typically accompanied by a figure that further illustrates the “problem state.” It is this problem for which the pattern provides a solution. Problem descriptions may also include common circumstances that can lead to the problem (also known as “forces”).

Solution

This represents the design solution proposed by the pattern to solve the problem and fulfill the requirement. Often the solution is a short statement followed by a diagram that concisely communicates the final solution state. “How-to” details are not provided in this section but are instead located in the *Application* section.

Application

This part is dedicated to describing how the pattern can be applied. It can include guidelines, implementation details, and sometimes even a suggested process.

Impacts

Most patterns come with trade-offs. This section highlights common consequences, costs, and requirements associated with the application of a pattern.

Note that these consequences are common but not necessarily predictable. For example, issues related to typical performance requirements are often raised; however, these issues may not impact an environment with an already highly scalable infrastructure.

Relationships

The use of design patterns can tie into all aspects of design and architecture. It is important to understand the requirements and dependencies a pattern may have and the effects of its application upon other patterns.

These diagrams are not exhaustive in that not all possible relationships a given design pattern can have are shown. Through the use of pattern relationship figures, this section merely highlights common relationships with an emphasis on how patterns support or depend on each other.

NOTE

Because there are two patterns in each relationship, almost every relationship is shown twice in this book: once in the *Relationships* section of each of the two patterns. To avoid content redundancy, most relationships are only described once. Therefore, if you find a relationship shown in a diagram that is not described in the accompanying text, refer to the description for the other pattern involved in that relationship. Note, however, that some relationships are considered self-explanatory and are therefore not described at all.

Details regarding the format of pattern relationship figures are provided in the *Pattern Notation* section earlier in this chapter.

Case Study Example

Most pattern profiles conclude with a case study example that demonstrates the sample application of a pattern in relation to the storylines established in Chapter 2.

SUMMARY OF KEY POINTS

- Each design pattern is documented with the same profile structure.
- Design pattern profiles begin with a requirements statement and an icon and then provide a summary table followed by sections with detailed descriptions.

5.5 Patterns with Common Characteristics

Each pattern in this book is distinct and unique and is considered an equal member of the overall pattern catalog. However, it is worth highlighting certain groups of similar patterns to better understand how they were named and why they share common characteristics.

NOTE

The following sections do not attempt to group patterns into formal categories. The upcoming chapters in Parts II, III, and IV already are subdivided by chapters representing specific pattern types. These sections here only point out that within and across these types, collections of patterns share common qualities and were labeled to reflect this.

Canonical Patterns

Canonical design patterns propose that the best solution for a particular problem is to introduce a design standard. The successful application of this type of pattern results in a canonical convention that guarantees consistent design across different parts of an inventory or solution.

The canonical design patterns in this book are:

- Canonical Protocol (150)
- Canonical Schema (158)
- Canonical Expression (275)
- Canonical Resources (237)
- Canonical Versioning (286)

Centralization Patterns

Centralization simply means limiting the options of something to one. Applying this concept within key parts of a service-oriented architecture establishes consistency and fosters standardization and reuse and, ultimately, native interconnectivity.

The following centralization patterns are covered in the upcoming chapters:

- Logic Centralization (136)
- Metadata Centralization (280)
- Process Centralization (193)
- Rules Centralization (216)
- Schema Centralization (200)
- Contract Centralization (409)
- Policy Centralization (207)

A common characteristic across centralization patterns is a trade-off between increased architectural harmony and increased governance and performance requirements. As explained shortly in the *Measures of Pattern Application* section, patterns can be applied to different extents. A key factor when assessing the application measure for centralization patterns is at what point the benefit outweighs the architectural impact.

NOTE

Centralization patterns are also very much related to the use of design standards. To constantly require that certain parts of a service-oriented architecture are centralized requires that supporting conventions be regularly followed.

SUMMARY OF KEY POINTS

- Canonical and centralization patterns need to be consistently applied to realize their benefits.
 - Canonical and centralization patterns require the use of supporting design standards.
-

5.6 Key Design Considerations

“Enterprise” vs. “Enterprise-wide”

Having discussed the notion of services as enterprise resources back in Chapter 4, it is important that there is a clear distinction between something that exists as a resource as part of an enterprise and something that is actually an *enterprise-wide* resource.

- An enterprise resource is not a resource that is necessarily made available across the entire enterprise. Instead, it is a resource positioned for use within the enterprise, outside of and beyond any one particular application boundary. In other words, it is a “cross-silo” resource.
- An enterprise-wide resource, on the other hand, is truly intended for use across all service inventories within an enterprise.

This difference in terminology is especially relevant to design patterns associated with specific enterprise boundaries, such as Domain Inventory (123). Note also that a service positioned as an enterprise resource is expected to be an inventory-wide resource, meaning that it is interoperable from anywhere within the inventory boundary.

Design Patterns and Design Principles

Most of the upcoming design patterns reference design principles where appropriate to highlight a dependency or relationship or perhaps to describe the effect a design pattern may have on service-orientation.

Specifically, the relationship between service-orientation design principles and patterns can be defined as follows:

- Design principles are applied collectively to solution logic in order to shape it in such a manner that it fosters key design characteristics that support the strategic goals associated with service-oriented computing.

- Design patterns provide solutions to common problems encountered when applying design principles—and—when establishing an environment suitable for implementing logic designed in accordance with service-orientation principles.

In many ways, design principles and patterns are alike. Both provide design guidance in support of achieving overarching strategic goals. In fact, it would not be unreasonable to think of the eight service-orientation principles as super patterns that are further supported by the patterns in this book.

Service-orientation design principles have another role in that they collectively define service-orientation as a design paradigm. Ultimately, it is best to view design patterns as providing support for the realization of design principles and their associated goals. (Design principles were introduced in the *Principles of Service-Orientation* section in Chapter 4.)

NOTE

We just stated that design principles could be thought of as super patterns. Why then weren't they documented as such? When the manuscript for this book was undergoing a review by Ralph Johnson and his pattern review group at UIUC, the question came up as to how to determine whether something is a legitimate pattern.

Ralph responded by stating, "When people ask me, 'Is this a pattern?' I usually say, 'That is not the right question.' The right question is whether pattern form is the best way to communicate this material." This is a good way to think of the purpose of this book.

Each pattern provides a specific solution to a distinct problem. The guidance provided by a design principle is much broader and can, in fact, end up solving a variety of problems. Therefore, design principles are better off documented in their form.

Design Patterns and Design Granularity

Design granularity, as it pertains to service-orientation, is itself something worth being familiar with prior to reading the upcoming chapters. Provided here are brief descriptions of common granularity-related terms:

- *Service Granularity* – The overall quantity of functionality encapsulated by a service determines the service granularity. A service's granularity is set by its functional context, which is usually established during the service modeling phase.
- *Capability Granularity* – The quantity of functionality encapsulated by a specific service capability determines the level of corresponding capability granularity.

- *Data Granularity* – The quantity of data exchanged by a specific service capability determines the level of its data granularity.
- *Constraint Granularity* – The extent of validation logic detail defined for a given service capability within the service contract determines the capability's level of constraint granularity. Generally, the more specific the constraints and the larger the amount of constraints, the more fine-grained the capability's constraint granularity is.

The effect of design patterns on service-related design granularity can vary. For example, when applying multiple patterns (or compound patterns) to the same service, the end-levels of design granularity may be distinctly defined by that combination of patterns (and they may fluctuate between the application of one pattern to another).

Measures of Design Pattern Application

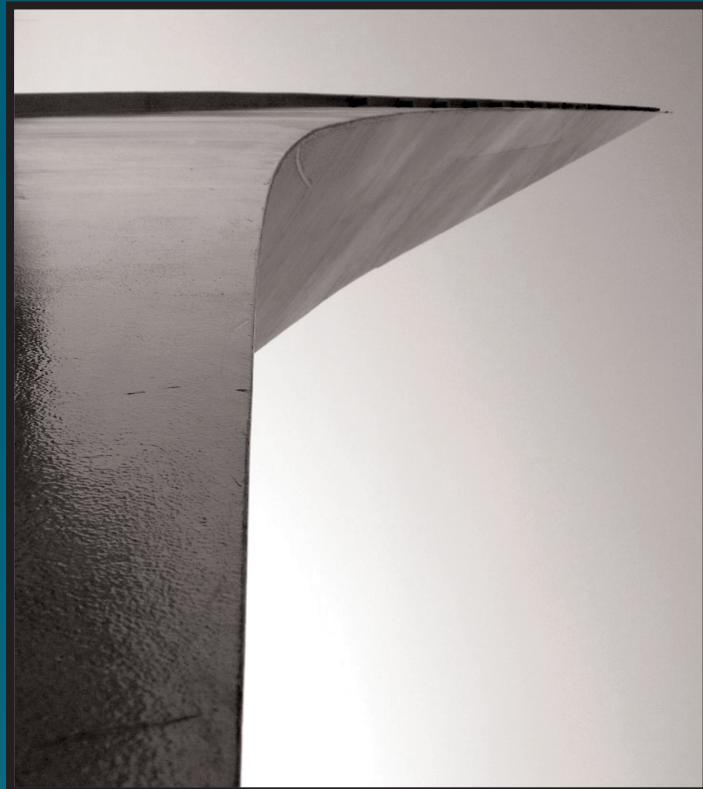
It is important to acknowledge that most patterns do not propose a black or white design option. Design patterns can often be applied at different levels. Although the effectiveness of a given pattern will generally be equivalent to the extent to which it is realized, there may be practical considerations that simply limit the degree to which a pattern can be applied in the real world (as is often the case when designing service logic that is required to encapsulate legacy functionality).

This consideration affects both design patterns and design principles. For example, individual service-orientation design principles can rarely be applied to their maximum potential. The point is to pursue the design goals of a design pattern or principle to whatever extent feasible and to strive for an end-result that realizes the pattern or principle to a meaningful extent.

SUMMARY OF KEY POINTS

- Some specific terminology is used within design pattern profiles. The distinction between “enterprise” and “enterprise-wide” is especially important.
 - Design pattern profiles contain references to related design principles, revealing links between the patterns and the realization of service-orientation itself.
 - As with design principles, most design patterns can be applied to various measures. Sometimes it isn't possible to fully apply a design pattern due to environmental constraints.
-

Part II



Service Inventory Design Patterns

Chapter 6: Foundational Inventory Patterns

Chapter 7: Logical Inventory Layer Patterns

Chapter 8: Inventory Centralization Patterns

Chapter 9: Inventory Implementation Patterns

Chapter 10: Inventory Governance Patterns

NOTE

The chapters so far have provided introductory content in preparation for the SOA design pattern catalog that is about to be covered within the next 16 chapters. Due to the unique documentation style required for a design pattern catalog, we will be changing a few formatting conventions at this point:

- The section numbers (1.x, 2.x, etc.) displayed throughout the first five chapters will be used sparingly in the remainder of the book. You will notice them in Chapters 6 and 11, but they are not used in any other remaining chapters.
- Unlike regular sections that flow sequentially across pages, each pattern profile section begins on a new page.
- Colored tabs are displayed on the edges of the pages within Chapters 6-22. These are designed to correspond to the colored tab legend located on the back cover of the book for quick reference purposes.

Additional style and formatting conventions that apply to the book in its entirety are explained in the *Symbols, Figures, Style Conventions* section in Chapter 1.



Chapter 6

Foundational Inventory Patterns

Enterprise Inventory

Domain Inventory

Service Normalization

Logic Centralization

Service Layers

Canonical Protocol

Canonical Schema

The patterns in this chapter are fundamental to defining a service-oriented architectural model with an emphasis on service inventory architecture. The design problems solved by these patterns help structure the architecture for the sole purpose of establishing a flexible and agile environment suitable for solution logic designed in accordance with service-orientation.

This chapter is structured so that the patterns are organized into a proposed application sequence. It is important to acknowledge that you are not required to follow this recommended sequence and that these patterns exist individually as part of the master pattern language provided by the book as a whole. You can therefore consider this chapter a “mini” structured pattern language that is part of a greater open-ended pattern language. As such, these patterns can be combined with patterns from other chapters into a variety of creative sequences.

As shown in Figure 6.1, the upcoming patterns are organized into the following groups:

1. *Inventory Boundary Patterns* – The scope of an architecture is defined by identifying the boundary of its corresponding service inventory and related characteristics.
2. *Inventory Structure Patterns* – The high-level structure and the overall complexion of the inventory itself is determined, which further influences the requirements that the eventual architecture implementation will need to fulfill.
3. *Inventory Standardization Patterns* – Key inventory-wide design standards are established to ensure a baseline level of service interoperability.

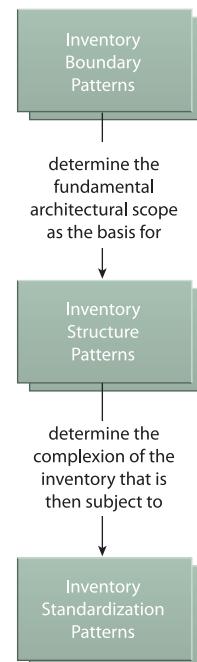


Figure 6.1

The patterns in this chapter are organized into groups that are ordered into a proposed application sequence in support of defining a layered inventory architecture.

The design patterns within a given group are explained at the beginning of each of the upcoming sections.

NOTE

Throughout this chapter you'll notice references to "agnostic" services and logic. If this term is new to you, skip ahead to the *Agnostic Logic and Non-Agnostic Logic* section at the beginning of Chapter 7 for an explanation. Also if you are not familiar with the term "service inventory," be sure to read the definition provided in Chapter 3.

How Inventory Design Patterns Relate to SOA Design Characteristics

It is interesting to note that the four fundamental design characteristics of a service-oriented architecture (established back in Chapter 4) are closely related to the application of the design patterns in this chapter, as shown in Table 6.1. The nature of these relationships will become clear when reading the pattern descriptions.

	Business Driven	Vendor Neutral	Enterprise Centric	Composition Centric
Enterprise Inventory	x	x	x	x
Domain Inventory	x	x	x	x
Logic Centralization	x		x	x
Service Normalization				x
Service Layers	x		x	x
Canonical Schema			x	x
Canonical Protocol			x	x

Table 6.1

How fundamental inventory design patterns (left) relate to the required base characteristics (top) of service-oriented architecture.

How Foundational Inventory and Service Patterns Relate

As mentioned earlier, this chapter provides a structured collection of patterns provided in a recommended application sequence. Chapter 11 provides a similarly organized set of patterns arranged in a suggested application sequence focused solely on service design. The fact that this chapter precedes Chapter 11 is not meant to indicate that the process established by this pattern application sequence should be completed prior to service design. Inventory design patterns are simply covered first in this book because they tend to be broader and coarser-grained in nature than those specific to service design.

In most modern SOA methodologies, iterative modeling and design processes are used to allow the design of an inventory architecture to be accomplished concurrently with service modeling and identification so that the definition of one can lead to the refinement of the other. It is completely up to you as to how these pattern groups are used in incorporated into project delivery cycles.

How Case Studies are Used in this Chapter

It's been a few chapters since we introduced the case study backgrounds, so it's worth a reminder that each of the upcoming design patterns is supplemented with a brief case study example. For this particular chapter, all examples relate back to the Alleywood Lumber Company environment established in the *Case #2 Background: Alleywood Lumber Company* section in Chapter 2.

As described in that chapter, Alleywood is facing an overhaul of their existing environment. An enterprise SOA initiative will require them to produce a collection of services that continue to represent their existing business lines, while also providing native inter-connectivity with existing Tri-Fold services. By progressing through the upcoming sequence of patterns, Alleywood's service-oriented technology architecture and a preliminary service inventory are defined.

6.1 Inventory Boundary Patterns

As explained in Chapters 3 and 4, a service inventory represents an independently standardized and governed collection of related services, and each such inventory is supported by its own, individual service-oriented technology architecture implementation. Therefore, a fundamental step in the creation of any service inventory is the definition of its scope in relation to the enterprise within which it resides.

The design patterns Enterprise Inventory (116) and Domain Inventory (123) shown in Figure 6.2 establish the boundary of the service inventory. Only one of these two patterns can be applied within a given IT enterprise.

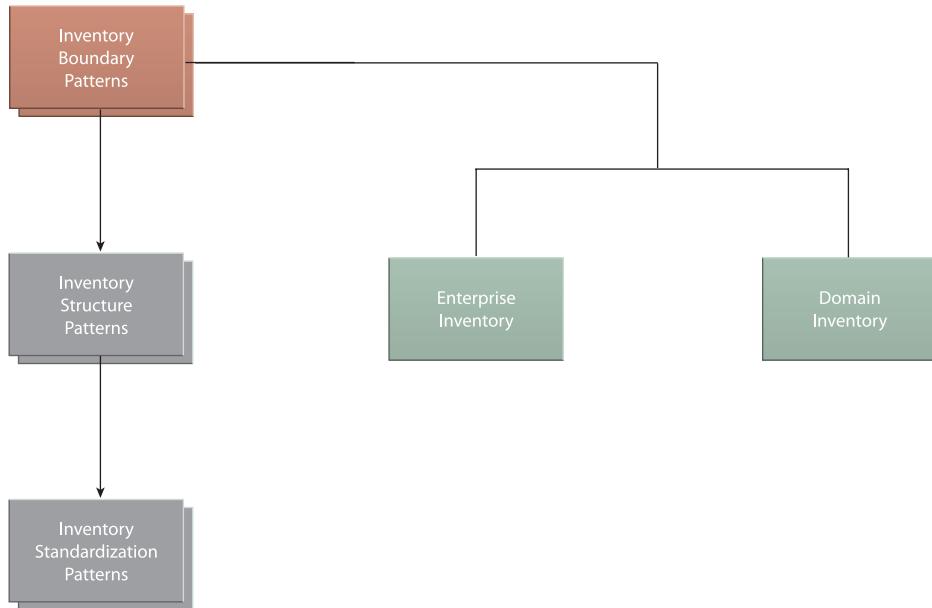


Figure 6.2

These inventory boundary patterns establish well-defined perimeters that determine the physical scope of an inventory.

Enterprise Inventory

How can services be delivered to maximize recombination?



Problem	Delivering services independently via different project teams across an enterprise establishes a constant risk of producing inconsistent service and architecture implementations, compromising recombination opportunities.
Solution	Services for multiple solutions can be designed for delivery within a standardized, enterprise-wide inventory architecture wherein they can be freely and repeatedly recomposed.
Application	The enterprise service inventory is ideally modeled in advance, and enterprise-wide standards are applied to services delivered by different project teams.
Impacts	Significant upfront analysis is required to define an enterprise inventory blueprint and numerous organizational impacts result from the subsequent governance requirements.
Principles	Standardized Service Contract, Service Abstraction, Service Composability
Architecture	Enterprise, Inventory

Table 6.2

Profile summary for the Enterprise Inventory pattern.

Problem

Throughout an enterprise, services can be delivered as part of various on-going development projects. Because each project has its own priorities and goals, services and supporting implementation architectures can easily be designed in isolation, optimized to fulfill tactical requirements.

The result is a collection of potentially disparate service clusters and technology architectures. The differences in these implementation environments can lead to serious problems when attempting to compose services into new configurations that span the initial architectural boundaries (Figure 6.3).

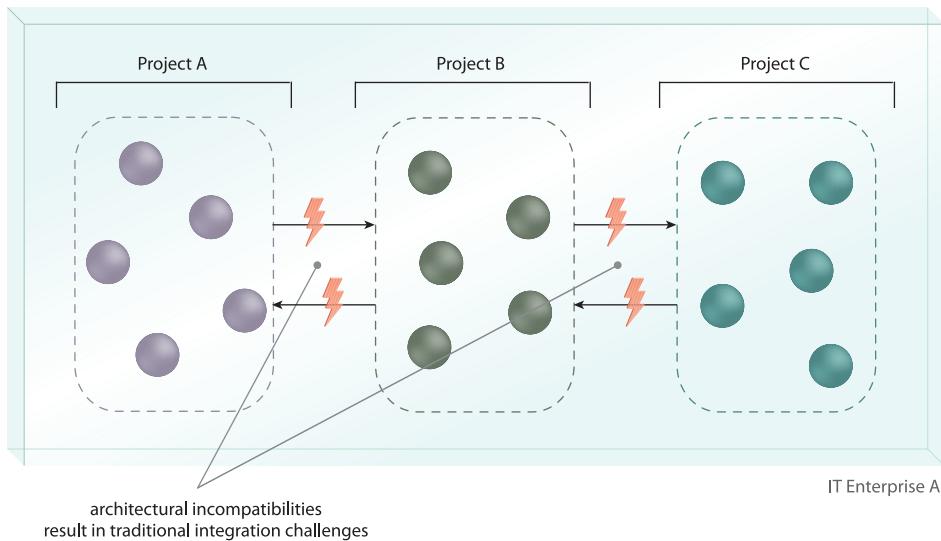


Figure 6.3

All services are built with the same vendor platform, but they are delivered via separate projects without taking into account a common architectural boundary. The end-result is an environment reminiscent of a silo-based enterprise that relies on constant integration effort to enable interoperability.

Solution

A service-oriented enterprise architecture is established to form the basis for a single enterprise service inventory. Services delivered as part of any project are designed specifically for implementation within the enterprise inventory's supporting architecture, guaranteeing wide-spread standardization and intrinsic interoperability.

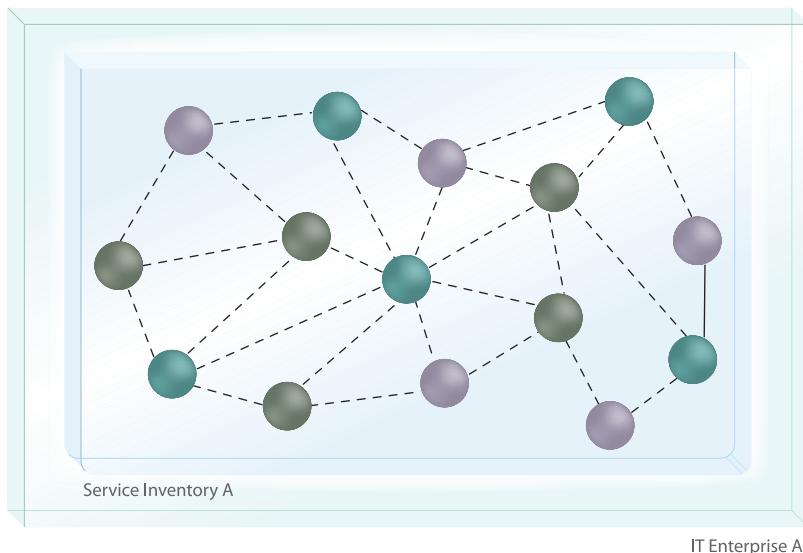


Figure 6.4

An enterprise service inventory establishes an enterprise-wide architectural boundary that promotes native interoperability and recomposition among all services.

Application

If the planned enterprise service inventory is significant in scope, then the organization needs to ensure it is capable of carrying out the corresponding SOA adoption effort.

Various factors come into play, each of which may introduce the need to reduce the scope or explore an alternative approach:

- the maturity of available technology for the planned services (especially for services being positioned as highly reusable enterprise resources)
- the maturity of governance technology platforms required to manage and evolve the service inventory as it is being built and after it is in place
- the order of magnitude associated with the amount of change and disruption brought on by the adoption
- the amount of legacy environments that are expected to constrain service encapsulation

- the financial resources required to carry out the adoption
- cultural and political obstacles that may arise as a result of the proposed changes and the required standardization effort

Therefore, this pattern is recommended for the following types of environments:

- small-to-medium-sized organizations with sufficient resources
- medium-to-large-sized organizations with highly controlled IT environments, a history of enterprise-wide standardization, or with the cultural flexibility to successfully adopt the required level of standardization
- medium-to-large-sized organizations that have the resources to build an enterprise service inventory while concurrently operating and maintaining their existing legacy systems
- new organizations that have no legacy systems and no IT history (and can therefore build an IT enterprise with a clean slate)

An enterprise service inventory does not need to encompass an *entire* enterprise. The purpose of this pattern is to establish a single service inventory with a scope sufficiently meaningful to warrant its creation.

Furthermore, the application of this pattern does not result in the creation of physical services. It establishes the concept of a service inventory on an enterprise-wide scale, for which services are conceptually defined through a planning and analysis effort that ties into the definition of a service inventory blueprint. To accomplish all of this typically requires a top-down analysis project that is completed by iteratively carrying out the service-oriented analysis and service modeling processes.

The following steps provide a suggested process:

1. Carry out planning and analysis stages to determine a preliminary scope for the service inventory that appears to be manageable based on the previously listed factors.
2. Collect all of the necessary enterprise business specifications that document business models and requirements that fall within the scope of the planned inventory (as well as those that are enterprise-wide). These specifications can include business entity models, logical data models, ontologies, taxonomies, business process definitions, and numerous other information and business architecture documents.

3. Using the enterprise-wide business models collected in Step 2, apply Entity Abstraction (175) and Process Abstraction (182) to establish a base set of business service layers.
4. Carry out the service-oriented analysis process iteratively by decomposing business process definitions that fall within the scope of the planned inventory. This results in the definition of service candidates that are continually refined. (See SOAMethodology.com for an example of a top-down process that iterates through the service-oriented analysis stage.)

NOTE

Several of the previously listed steps also apply to the upcoming pattern Domain Inventory (123). The primary distinctions are scope and quantity. The ultimate goal of Enterprise Inventory is to establish a *single* service inventory that spans as much of the IT enterprise as possible, whereas Domain Inventory (123) allows for multiple (usually smaller) inventories to exist within an enterprise.

Impacts

To achieve unity across an enterprise-wide service inventory, a large (and sometimes monumental) amount of top-down analysis may be required so that service candidates can be modeled and aligned with each other prior to their actual delivery. This can lead to an expensive and time-consuming up-front analysis project.

Alternative methodologies can be employed to phase in the delivery of services with less initial analysis. One example is the “meet-in-the-middle” approach that allows for analysis to occur on an on-going basis while services are built and implemented. There is then a commitment to “re-align” the services at a later point after the analysis produces a mature enterprise-wide inventory blueprint. Although a proven strategy that overcomes the time-burden of top-down approaches, this method can introduce additional complexities and increased expense.

Common issues that challenge the creation of an enterprise service inventory are documented in the *Problem* section of the pattern description for Domain Inventory (123) because this pattern provides an alternative approach that directly addresses concerns associated with Enterprise Inventory.

Relationships

Enterprise Inventory establishes an architectural boundary with a physical structure that is further subject to the application of a series of additional inventory-related patterns. Inventory Endpoint (260) in particular can complement this pattern by providing standardized access to consumers outside the enterprise. Domain Inventory (123) provides the primary alternative to this pattern.

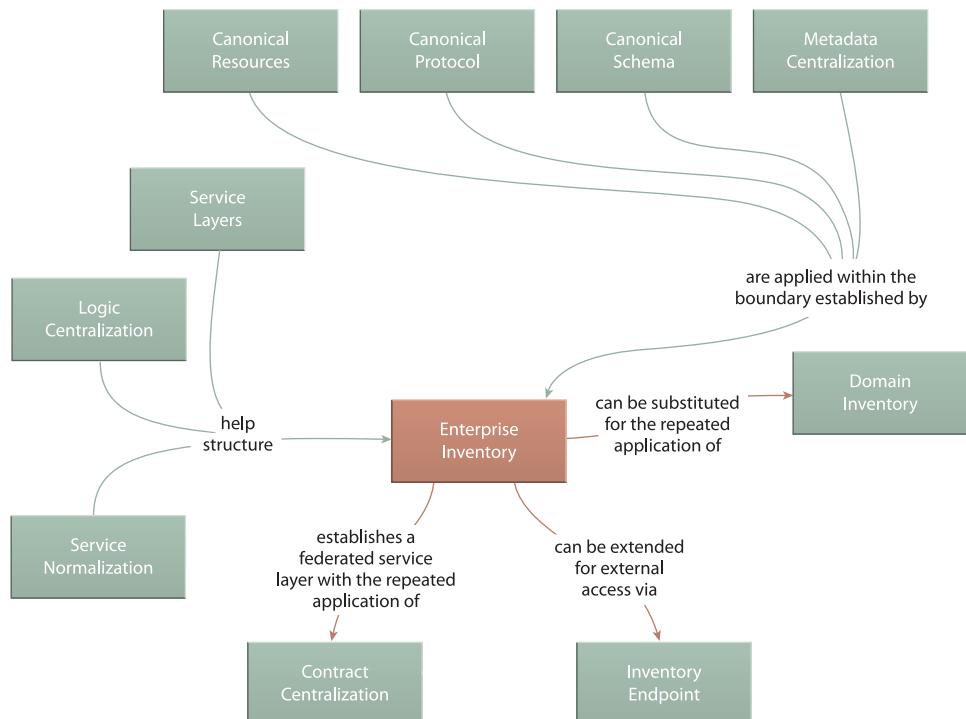


Figure 6.5

Enterprise Inventory determines scope, but it relies on other patterns to establish the inventory structure.

CASE STUDY EXAMPLE

Armed with enterprise architecture standards that dictate technical constraints and further establish a centralized, Web services-centric communications framework, Alleywood and Tri-Fold teams have been able to incorporate strategic business requirements and goals to define a target inventory architecture.

To further set its physical boundary, additional analysis is required to determine what logic the planned service inventory will actually be required to encompass. Business analysts and subject matter experts are employed to participate in a series of service-oriented analysis and service modeling processes where existing and extended business processes are studied and used as a source from which to derive a variety of services.

These processes are carried out iteratively to produce a service inventory blueprint for the collection of services Alleywood is planning to deliver. By performing these processes and applying the service definition patterns described in Chapter 11, a set of initial service candidates is defined (Figure 6.6).

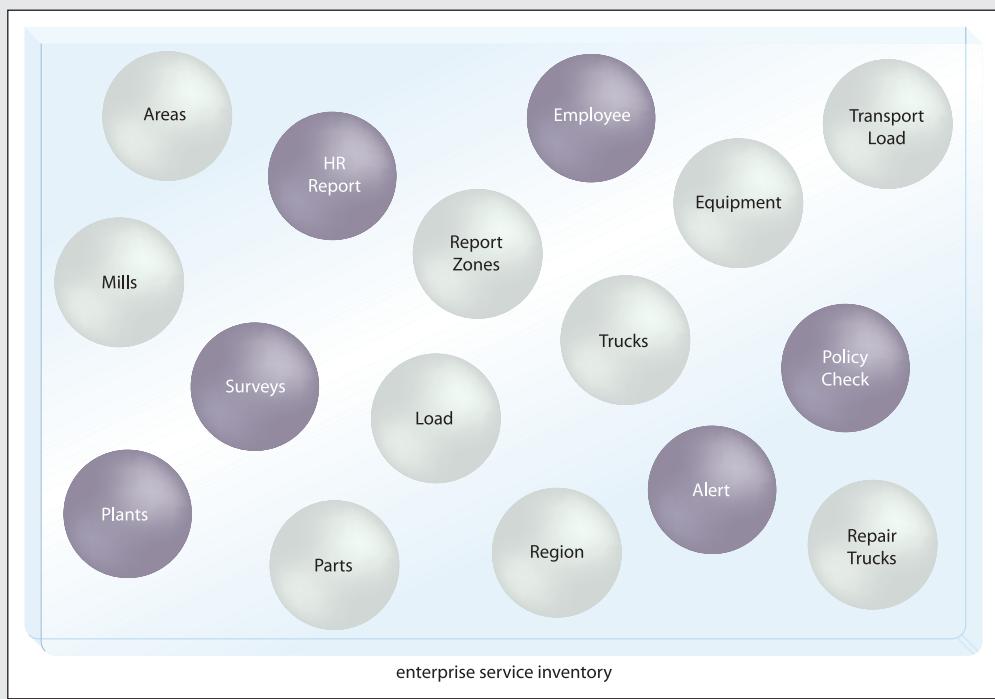
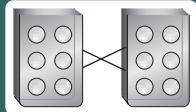


Figure 6.6

The initial set of services that comprise an enterprise service inventory that spans Alleywood and Tri-Fold environments. Gray services belong to Alleywood while purple-colored services originated with Tri-Fold.

Domain Inventory

How can services be delivered to maximize recombination when enterprise-wide standardization is not possible?



Problem	Establishing a single enterprise service inventory may be unmanageable for some enterprises, and attempts to do so may jeopardize the success of an SOA adoption as a whole.
Solution	Services can be grouped into manageable, domain-specific service inventories, each of which can be independently standardized, governed, and owned.
Application	Inventory domain boundaries need to be carefully established.
Impacts	Standardization disparity between domain service inventories imposes transformation requirements and reduces the overall benefit potential of the SOA adoption.
Principles	Standardized Service Contract, Service Abstraction, Service Composability
Architecture	Enterprise, Inventory

Table 6.3

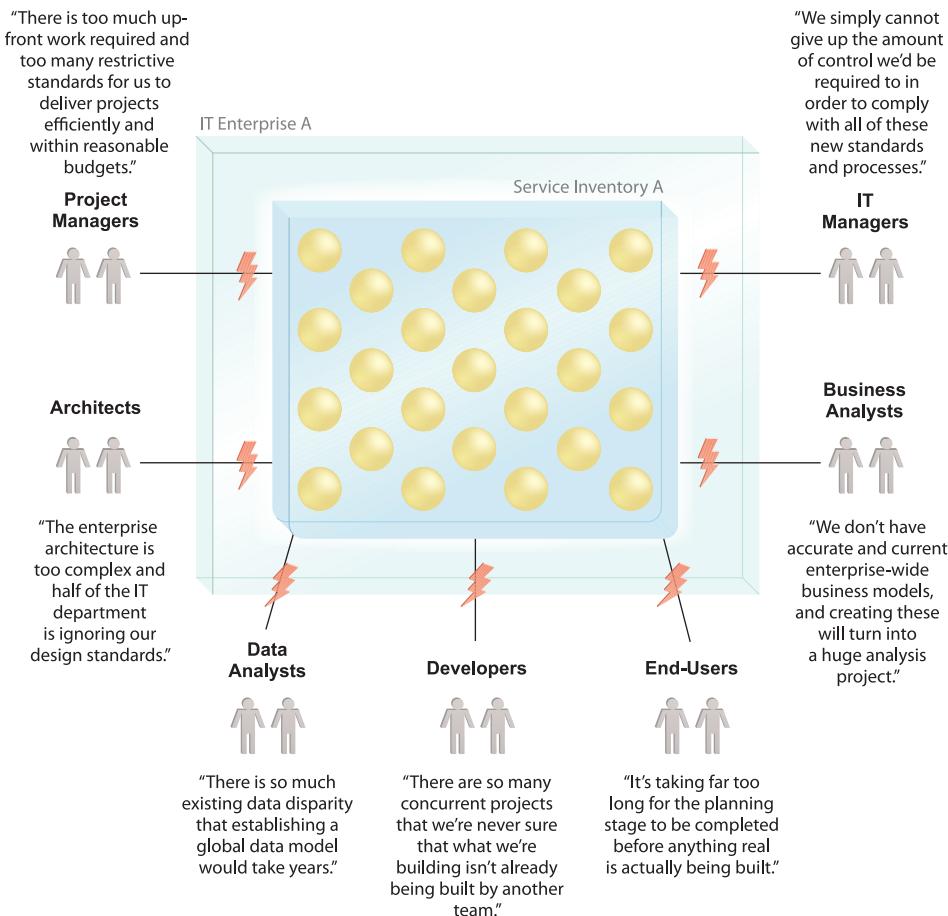
Profile summary for the Domain Inventory pattern.

Problem

In larger environments it can be impractical or even unrealistic to define and maintain a single service inventory for an entire enterprise. Standardization and governance issues can raise numerous concerns, most of which tend to be organizational in nature (Figure 6.7).

NOTE

Several of the issues raised in Figure 6.7 relate to the governance of SOA projects and implementations. Organizational and technology-related governance topics (and patterns) will be covered separately in the upcoming title *SOA Governance* as part of this book series.

**Figure 6.7**

Common organizational issues that hinder efforts to establish a single enterprise service inventory.

Solution

Multiple service inventories are created for one enterprise. The scope of each represents a well-defined enterprise domain. Within domains, service inventories are standardized and governed independently (Figure 6.8).

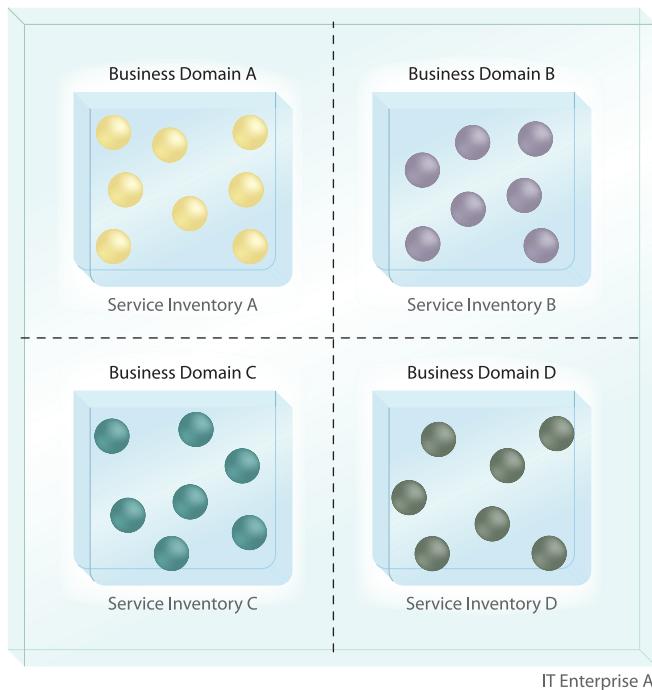


Figure 6.8

An enterprise partitioned into domain service inventories, each representing a pre-defined domain.

Application

Whether or not to apply this pattern is tied to the question of whether an enterprise service inventory is feasible within a given environment. Many factors (most of which are specific to the organization) weigh in on this decision point. However, some general guidelines are available.

For example, domain service inventories are an appropriate alternative when any of the following supporting factors exist:

- The implementation environment is a large enterprise without strong executive sponsorship and wide-spread support for the SOA initiative.
- The enterprise does not have an established, global data models and creating them is considered unrealistic.
- The organization is incapable of changing the complexion of its IT departments in support of a more centralized governance model.

The application of this pattern can bypass these obstacles and accelerate the transition toward service-orientation in that an enterprise-level migration can be delivered in phases that each result in the creation of a manageable collection of services. For many organizations, this pattern provides the only realistic option for adopting SOA.

The key to creating effective domain inventories is to clearly define the domains in advance, thereby establishing a sub-divided view of an enterprise prior to building any one inventory.

Organizations will often have options as to how the domains are defined. Here are some common examples:

- Organizational business areas represented by specific IT departments or groups. These business areas would then establish the basis for business domains.
- Organizational business domains not represented by separate IT departments or groups. These domains can still form the basis of service inventory boundaries but require cooperation across IT departments.
- Remote offices, each with its own IT department and development center. This can result in geographical-based domains.

Ideally, domain inventories correspond to enterprise business domains, such as those based on an organization's lines of business. This allows each inventory to be tuned to and evolve with its corresponding set of business models in full support of establishing the business-driven architecture characteristic.

Impacts

Multi-domain service inventory implementations make some impositions, in that they allow for individual inventories to be standardized differently. This generally results in the need to introduce targeted transformation for cross-domain interoperability as part of the overall enterprise architecture.

The ultimate benefits associated with achieving a unified and federated enterprise service-oriented architecture are scaled back to whatever extent domain inventories are created.

Transformation requirements that emerge to enable cross-domain data exchange impact the development and design effort of corresponding service compositions and also add performance overhead to their runtime execution. Furthermore, the independence by which each inventory can be built and evolved will often lead to the creation of redundant services across domains.

Relationships

The same design patterns that structure an enterprise inventory will end up structuring an inventory defined via Domain Inventory (only the scope will be smaller). However, unlike Enterprise Inventory (116), the application of this pattern will generally result in the need for transformation patterns, such as Protocol Bridging (687) and Data Model Transformation (671). Inventory Endpoint (260) will also play a more prominent role to facilitate cross-inventory communication.

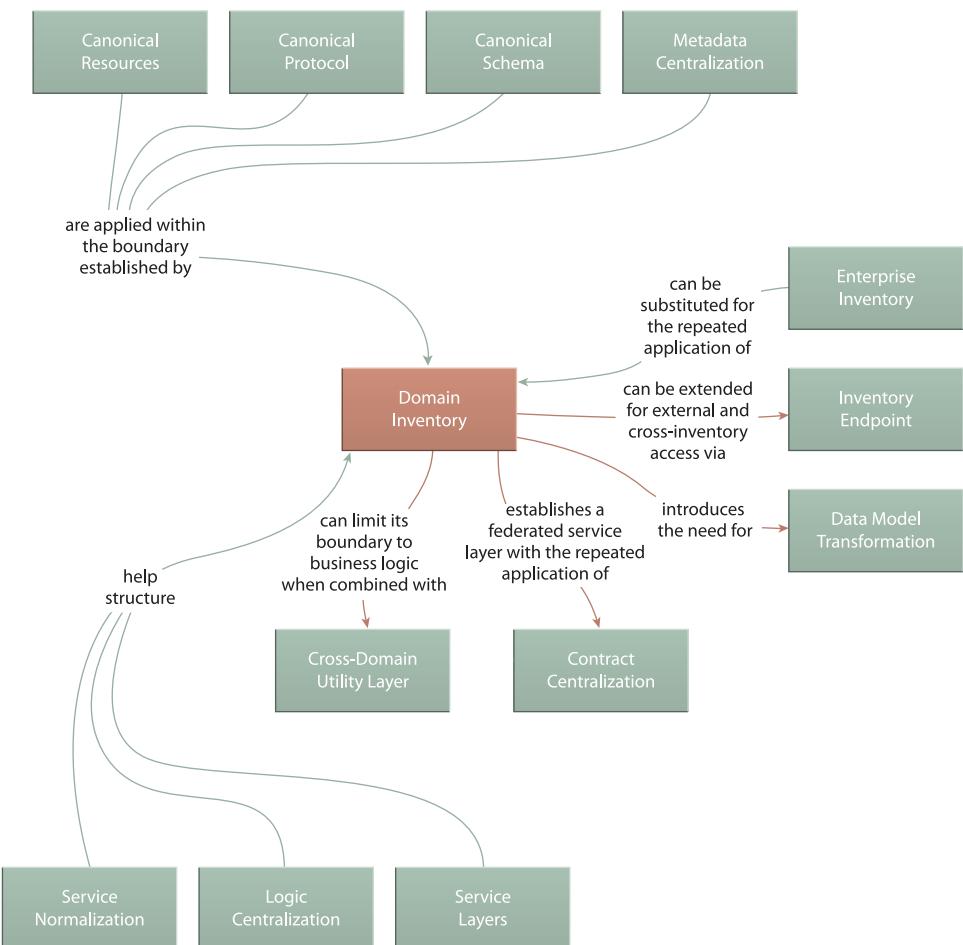


Figure 6.9

Domain Inventory shares many of the same relationships as Enterprise Inventory (116) but introduces new requirements that can be fulfilled by additional patterns more specific to a domain-based environment.

CASE STUDY EXAMPLE

The original goal of McPherson enterprise architects was to establish a global collection of services that would span Alleywood and Tri-Fold platforms. Although it was understood that this would require an unprecedented standardization and federation effort, the strategic benefits seemed to justify it.

However, subsequent to the creation of a more detailed transition plan and associated metrics and cost estimates, many further discussions with management and key IT personnel within Alleywood, Tri-Fold, and other McPherson departments resulted in the identification of several concerns:

- The cost of the standardization effort is much higher than expected, primarily due to the outstanding data architectures that need to be defined.
- Incompatibilities have been identified between some of the preliminary data remodeling that has been performed by Alleywood and the schemas that have already been implemented as part of Tri-Fold's ERP environment. To bring these into alignment would require changing existing Tri-Fold service implementations.
- Alleywood management has complained about several design standards that have been imposed upon them in order to deliver services in compliance with Tri-Fold requirements. Due to the difference in implementation technology, Alleywood feels these standards introduce awkward and inefficient design characteristics into their planned service designs.
- Alleywood and Tri-Fold use different business modeling methodologies. This has resulted in different forms of business model specifications. Furthermore, meetings between analysts from each organization have been strained due to differences in philosophy and conventions associated with business analysis processes.

These and other reasons have prompted a change in the original strategic plan. Instead of creating a single pool of federated services, separate domain inventories will be established (Figure 6.10). This will allow Alleywood and Tri-Fold teams to design and govern their services independently from each other.

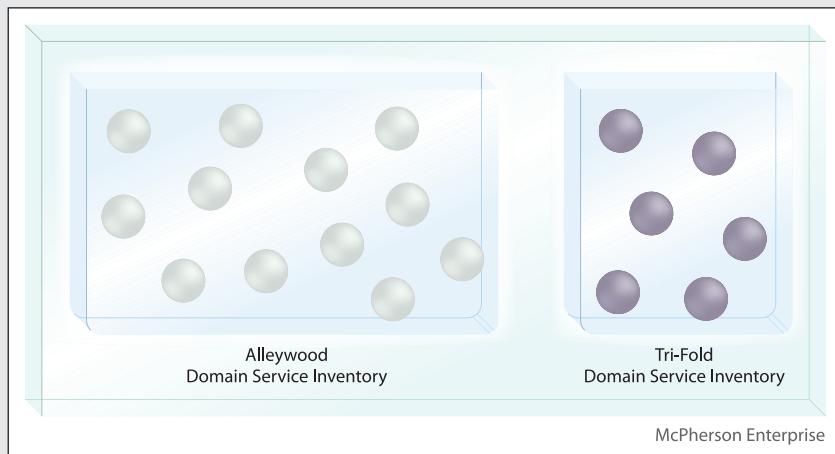


Figure 6.10

Alleywood and Tri-Fold establish physically separate service inventories. Note how the quantity of overall services increases when compared to the previous Enterprise Inventory (116) case study example due to the need to create redundant logic across the physical domains.

6.2 Inventory Structure Patterns

Once the inventory boundary is determined, the complexion and organization of the inventory itself needs to be determined on a fundamental level. The next set of patterns helps define the underlying inventory structure by establishing basic service boundaries and classifications.

As shown in Figure 6.11, Service Normalization (131) and Logic Centralization (136) influence the inventory structure by requiring that future services be in alignment with each other and that future agnostic services in particular be positioned as the sole endpoint for the logic they represent.

There is no proposed sequence within this pattern group. These patterns simply establish modeling and design parameters on an inventory-wide basis.

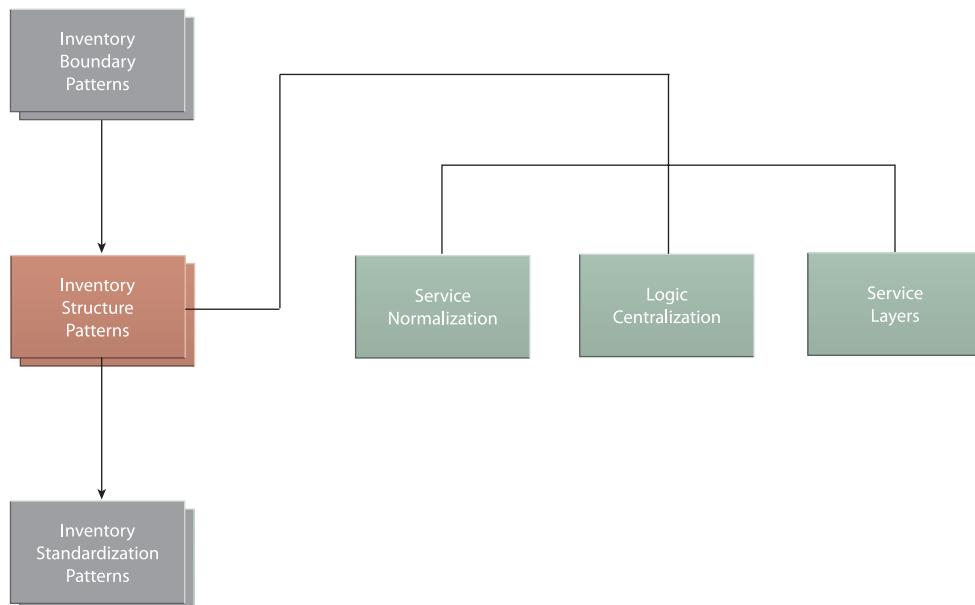
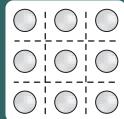


Figure 6.11

The inventory structure patterns align and position services within an inventory and further organize the inventory into logical abstraction layers.

Service Normalization

How can a service inventory avoid redundant service logic?



Problem	When delivering services as part of a service inventory, there is a constant risk that services will be created with overlapping functional boundaries, making it difficult to enable wide-spread reuse.
Solution	The service inventory needs to be designed with an emphasis on service boundary alignment.
Application	Functional service boundaries are modeled as part of a formal analysis process and persist throughout inventory design and governance.
Impacts	Ensuring that service boundaries are and remain well-aligned introduces extra up-front analysis and on-going governance effort.
Principles	Service Autonomy
Architecture	Inventory, Service

Table 6.4

Profile summary for the Service Normalization pattern.

Problem

The boundary of a service is defined by its functional context and the collective boundaries of its capabilities. Even within a pre-defined inventory boundary, when services are delivered by multiple project teams there is a risk that some will provide functionality that will overlap with others.

This leads to a denormalization of the inventory (Figure 6.12), which can cause several problems, such as:

- an inability to establish service capabilities as the official endpoints for bodies of agnostic logic
- a more convoluted architecture wherein services with overlapping functionality can become out of sync, providing the same functions in different ways

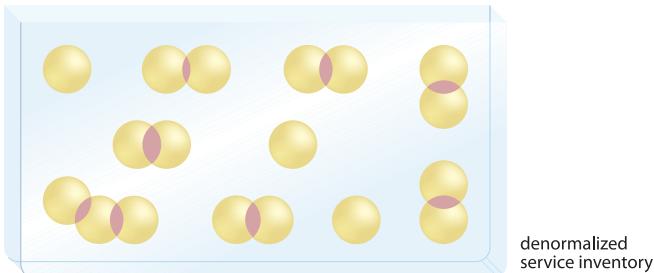


Figure 6.12

A service inventory containing services with overlapping functional boundaries that introduce denormalization.

Solution

Services are collectively modeled before their individual physical contracts are created. This provides the opportunity for each service boundary to be planned out so as to ensure that it does not overlap with other services. The result is a service inventory with a higher degree of functional normalization (Figure 6.13).

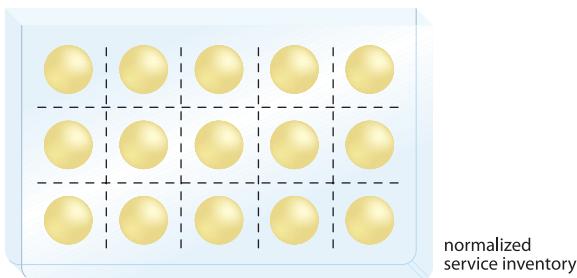


Figure 6.13

When services are delivered with complementary and well-aligned boundaries, normalization across the inventory is attained. Note also how the quantity of required services is reduced.

Application

The goals of this pattern are best realized by pursuing service-level autonomy (as associated with the Service Autonomy design principle) during the service modeling stage.

Common steps include:

1. Identifying and decomposing a business process definition that pertains to the inventory boundary.
2. Allocating the individual parts of the process into appropriate new or existing conceptual service candidates.

3. Validating that no two service boundaries overlap.

These steps are part of a greater service modeling process that takes other modeling considerations into account. To fully apply Service Normalization requires that this process be carried out iteratively, once for every business process that is associated with the scope of the service inventory. Through these iterations, the functional contexts and boundaries of service candidates are repeatedly refined and validated.

The end-result is a service inventory blueprint that provides a normalized view of all services within the inventory. This approach is typically part of a top-down delivery effort.

NOTE

The service modeling process explained in *Service-Oriented Architecture: Concepts, Technology, and Design* contains steps that address normalization issues. However, because larger modeling efforts may result in this process being carried out concurrently by different teams, a subsequent inventory blueprint-wide review is always recommended. Other approaches for achieving the goals of Service Normalization may also exist as part of different methodologies.

Impacts

The guarantee of inventory-wide normalization requires that all services be conceptually modeled prior to delivery, as part of an inventory service blueprint specification. Depending on the scope of the planned inventory, this can result in a separate analysis project that needs to be completed before any service can actually be built.

Continual governance effort is further required to ensure that services maintain normalization throughout the inventory as they are revised and evolved over time.

Relationships

Service Normalization lays the foundation for Contract Centralization (409) by ensuring that no two services share the same functionality. This allows contracts to be positioned as the sole entry point into service logic and further enables those services to be independently evolved, as per Service Refactoring (484). Schema Centralization (200) and Policy Centralization (207) further support this pattern by avoiding contract-related redundancy.

To successfully preserve a normalized inventory requires the consistent enforcement of Logic Centralization (136), making these two patterns very closely related.

Contract Denormalization (414) is referred to in Figure 6.14 only to indicate that, despite its name, it does not interfere with the goals of this pattern. As explained in Chapter 16, Proxy Capability (497) must violate this pattern out of necessity when services require post implementation decomposition.

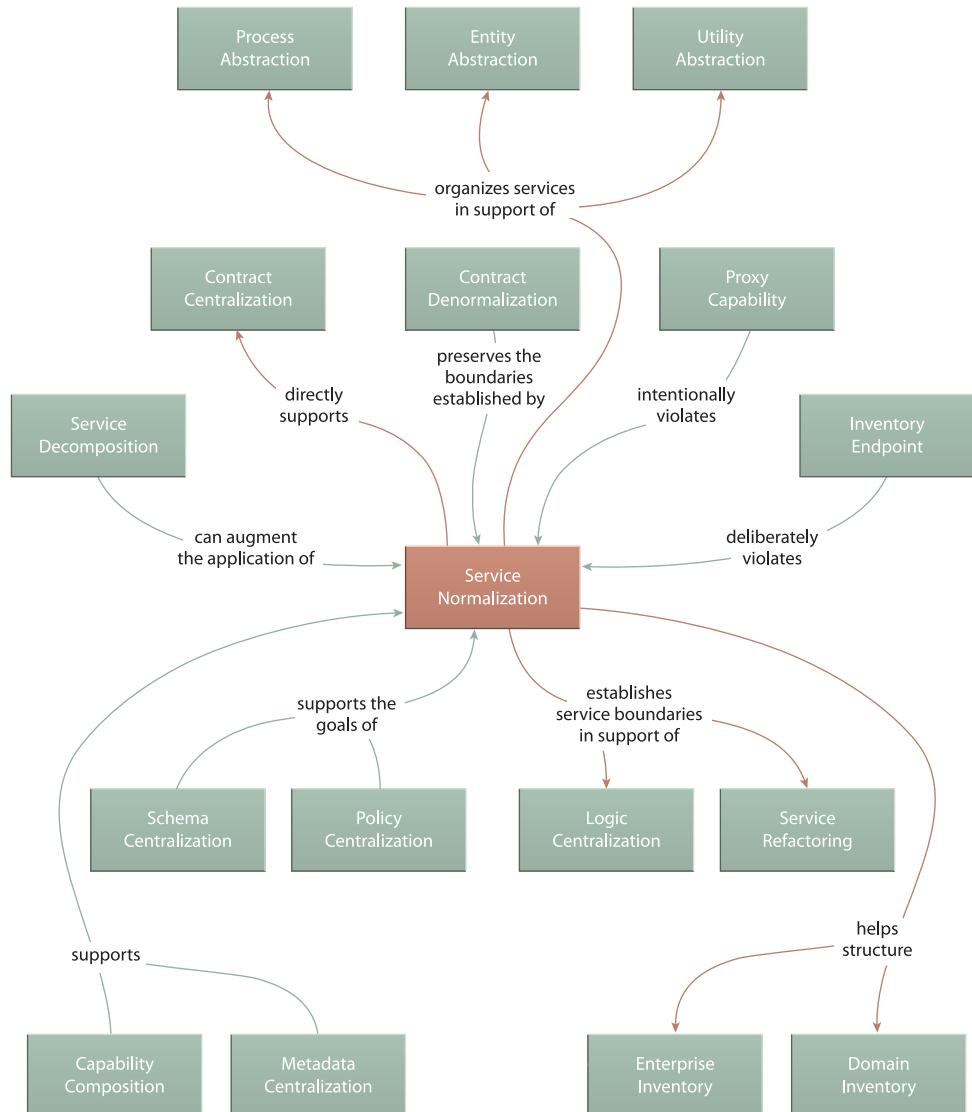


Figure 6.14

Service Normalization fundamentally organizes a service inventory but relies on the successful application of other patterns to retain this state.

CASE STUDY EXAMPLE

Because the initial version of the Alleywood inventory service blueprint was defined via a collaborative effort comprised of different groups of architects and business analysts delivering different service candidates, there is a risk that some of the proposed service contexts functionally overlap with others.

A blueprint-wide review is conducted to look for any potential denormalization of the established service boundaries. A few are detected and subsequently corrected by adjusting the parent contexts of the affected services. One such example is shown in Figure 6.15.

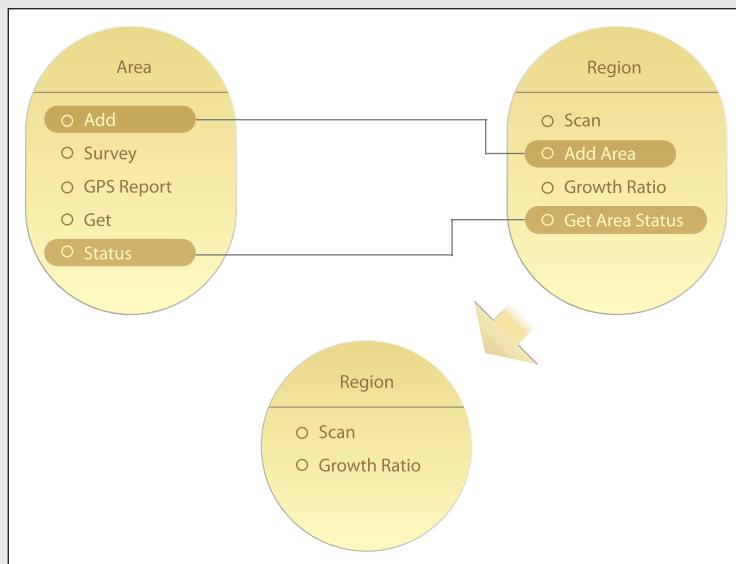
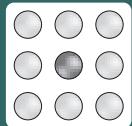


Figure 6.15

Those who modeled the Region service assumed it would encompass area-related processing. However, an Area service was also being delivered by a separate modeling team. As a result, the Region service capabilities associated with area record processing are removed to avoid functional overlap and increase overall inventory normalization.

Logic Centralization

How can the misuse of redundant service logic be avoided?



Problem	If agnostic services are not consistently reused, redundant functionality can be delivered in other services, resulting in problems associated with inventory denormalization and service ownership and governance.
Solution	Access to reusable functionality is limited to official agnostic services.
Application	Agnostic services need to be properly designed and governed, and their use must be enforced via enterprise standards.
Impacts	Organizational issues reminiscent of past reuse projects can raise obstacles to applying this pattern.
Principles	Service Reusability, Service Composability
Architecture	Inventory, Composition, Service

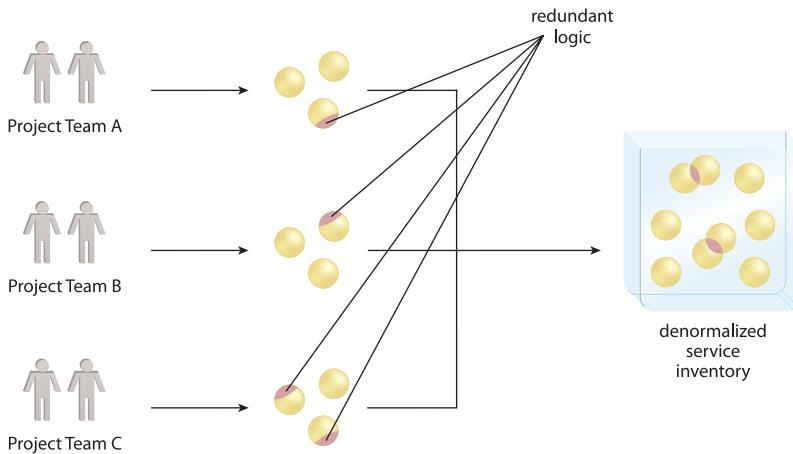
Table 6.5

Profile summary for the Logic Centralization pattern.

Problem

As we established in earlier chapters, reuse represents a key characteristic that typically needs to be realized on a broad scale for some of the more strategic goals associated with service-orientation to be attained. However, even if well-designed agnostic services are consistently delivered into a service inventory, it does not guarantee that project teams building new solutions will use them.

For various reasons, it may be easier, simpler, or just more practical to avoid involving reusable services in order to concentrate on the fulfillment of short-term, tactical delivery goals. This approach may be convenient, but it eventually results in a denormalized service inventory where functional redundancy is common (Figure 6.16).

**Figure 6.16**

Different project teams delivering services with redundant logic leads to functional overlap among services in the inventory.

Solution

To pursue the strategic goals associated with service reuse, the characteristic of reuse itself must form the basis of supporting internal design standards. The foremost of these standards needs to dictate that services classified as agnostic must become a primary (or even sole) means by which the logic they represent is accessed. This forms the basis for Logic Centralization, as depicted in Figure 6.17. The level to which the centralization of logic succeeds as an enterprise-wide standard determines the extent to which the repeated ROI of services can be realized.

Application

When services are built by different project teams, there is always the risk that one team will develop a service with new logic that exists as part of an already-implemented agnostic service.

Common reasons for this are:

- The project team is not aware of the agnostic service's existence or capabilities because the service is not sufficiently discoverable or descriptive.
- The project team refuses to use the existing agnostic service because it is considered burdensome to do so.

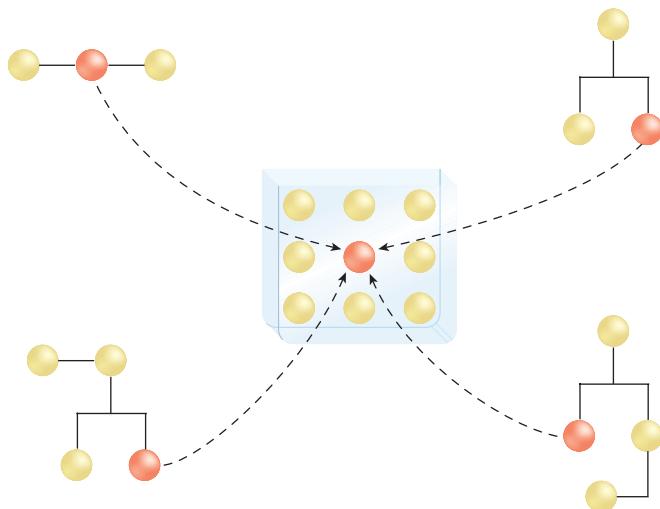


Figure 6.17

Service consumers are required to reuse functionality provided by a single designated agnostic service.

While the former scenario can be avoided through the application of Metadata Centralization (280), the latter is where an inventory-wide design standard is required. In fact, the manner in which this pattern is applied is through the creation and enforcement of a standard that requires that services act as the sole entry point for the functional boundaries they represent within a given inventory.

This type of standard essentially dictates that agnostic services must always be used as intended, even if they do not yet possess all required functions. For example, if a new capability needed by a project team clearly falls within the boundary of an existing service, the corresponding functionality needs to be added to that service instead of ending up elsewhere (Figure 6.18).

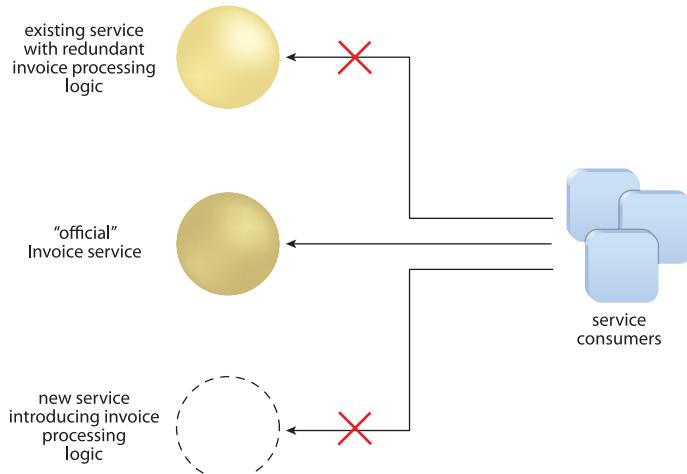


Figure 6.18

In this case, only one service is considered the “official” entry point for invoice-related processing.

Impacts

As straightforward as Logic Centralization may sound, it can be difficult to achieve, especially with broadly scoped service inventories. For larger organizations working toward an enterprise service inventory attaining a state where all development project teams agree to not build redundant logic and instead use existing services may seem like an unattainable ideal.

Introducing Logic Centralization into an organization that does not have a history of fostering reuse or using design standards in general will almost always raise cultural issues with people and IT departments affected by service delivery projects.

For example:

- Existing project plans and processes are impacted by requiring the involvement of reusable services as part of their development projects.
- There may be resistance to giving up control of solution designs if teams are forced to include existing agnostic services or produce new services that need to be reusable.

These concerns need to be addressed prior to the delivery of agnostic services to avoid compromising the strategic value of a service inventory. If only partial support for the delivery and usage of reusable services is received within an IT division, the risk of ending up with a denormalized and potentially convoluted inventory architecture is significant.

Relationships

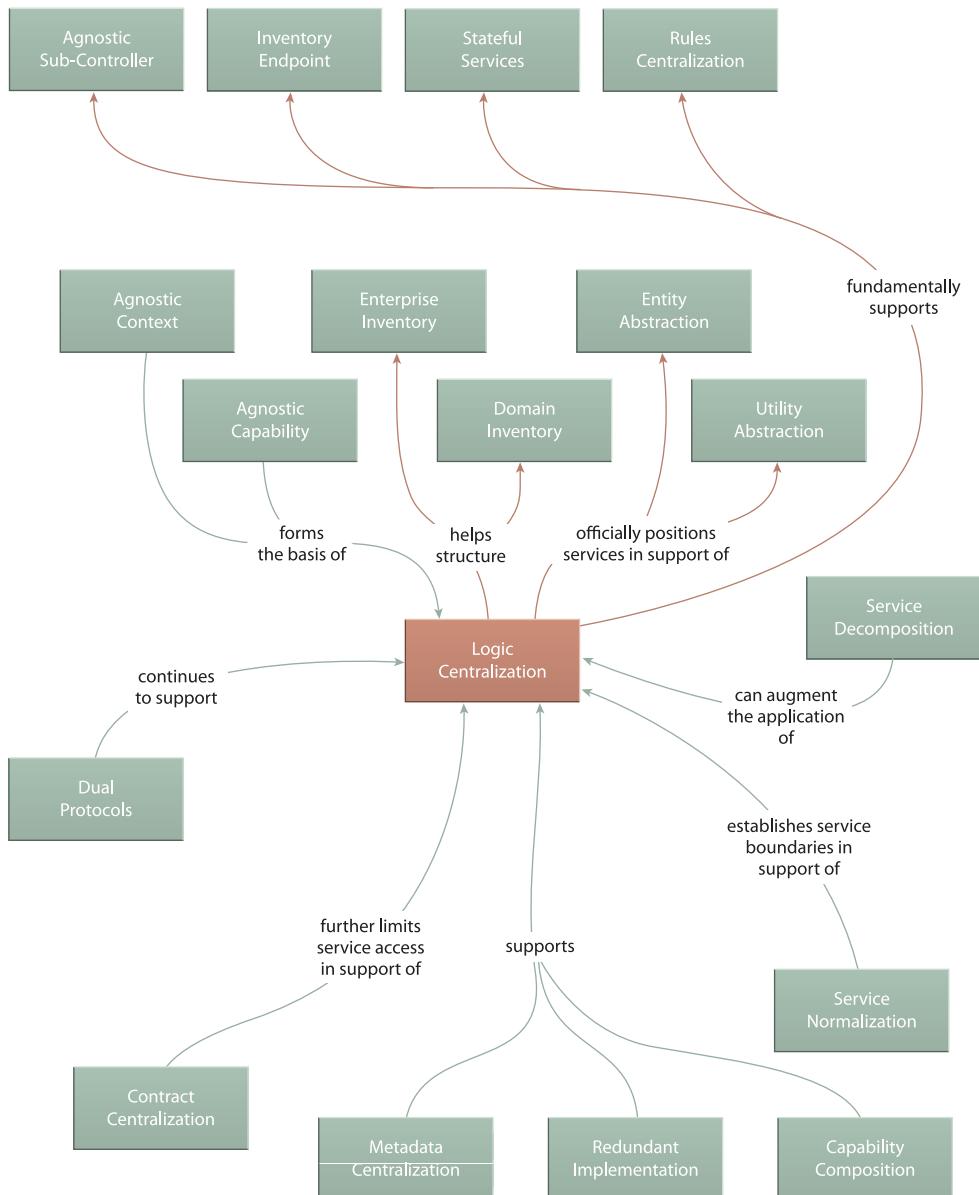
Logic Centralization is a core design pattern very much focused on centralizing *agnostic* logic, which is why it is commonly associated with Entity Abstraction (175) and Utility Abstraction (168) and also why its application is influenced by Agnostic Context (312) and Agnostic Capability (324).

Whereas Service Normalization (131) primarily solves a service modeling problem, Logic Centralization addresses service usage concerns. In a way, Logic Centralization helps attain the goals of Service Normalization (131).

Contract Centralization (409) also has a very close relationship with Logic Centralization because together they position official services that can only be accessed via official entry points (contracts), which is fundamental to establishing a healthy federated endpoint layer.

There are also numerous peripheral relationships with additional specialized patterns, such as Metadata Centralization (280), which supports the discovery of services to which Logic Centralization has been applied, and Redundant Implementation (345), which supports the scalability demands that tend to fall upon centralized services. Additional examples are shown at the top and bottom of Figure 6.19.

Perhaps its most important relationship is with Capability Composition (521), a pattern that introduces the fundamental rule that logic outside of a service's boundary must be composed.

**Figure 6.19**

Logic Centralization supports the goals of many design patterns but is itself also supported by others.

CASE STUDY EXAMPLE

The preceding Service Normalization (131) example introduced the Area and Region services that had been modeled concurrently and therefore resulted in the definition of overlapping functional boundaries. Alleywood architects speculate that if this overlap had not been corrected during the analysis phase, two services providing redundant functionality would have been delivered within the same service inventory.

Whenever Alleywood service consumer designers would have required the affected Area capabilities, they would have chosen to use either one of the two services, or they would have perhaps only discovered one of the two and simply used that one only. Either scenario risks the misuse of the Region service for area-related functions because no requirement exists to use one over the other.

To prevent this from ever happening, especially with subsequent service delivery projects that may not have the benefit of up-front modeling phases, a special enterprise standard is established. It essentially dictates that for any body of agnostic logic, only one official service is positioned as the endpoint. This standard applies even though the possibility of multiple endpoints will continue to exist.

In Alleywood's case, the enforcement of this standard is tied to the use of a service registry from where any project team can locate the official service for a particular type of capability. Therefore, Alleywood applies this pattern together with Metadata Centralization (280).

Service Layers

How can the services in an inventory be organized based on functional commonality?



Problem	Arbitrarily defining services delivered and governed by different project teams can lead to design inconsistency and inadvertent functional redundancy across a service inventory.
Solution	The inventory is structured into two or more logical service layers, each of which is responsible for abstracting logic based on a common functional type.
Application	Service models are chosen and then form the basis for service layers that establish modeling and design standards.
Impacts	The common costs and impacts associated with design standards and up-front analysis need to be accepted.
Principles	Service Reusability, Service Composability
Architecture	Inventory, Service

Table 6.6

Profile summary for the Service Layers pattern.

NOTE

This pattern should not be confused with Service Layer (Fowler, Stafford). Whereas the goal of Service Layers is to establish logical domains represented by collections of related services, the application of Service Layer results in an externally facing interface layer for a specific application.

Problem

Within a typical service inventory there will tend to be services that have similar functional contexts. However, these services may be designed and implemented differently, depending on the nature of the delivery project. This leads to a missed opportunity to establish consistency in how service boundaries are defined and in the nature of the logic they encapsulate. The result is an inventory of services that cannot easily (or cleanly) be partitioned into groups for the purpose of sub-domain based abstraction and governance (Figure 6.20).

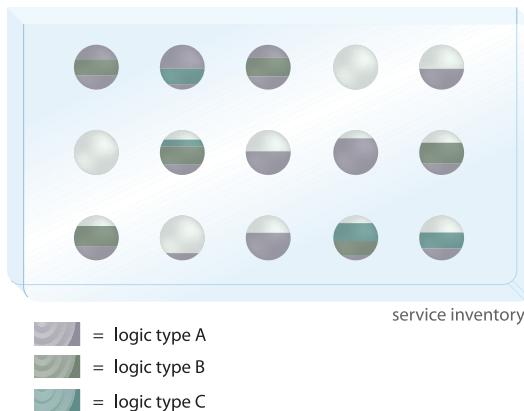


Figure 6.20

Arbitrarily delivered services exist as hybrids where many encapsulate different forms of logic. This can inhibit their reusability and long-term governance.

Solution

A service inventory architecture can formally establish classification profiles to represent common types of services within a given inventory. These profiles are referred to as *service models*, each of which represents a unique set of design characteristics associated with a well defined service category. Service models form the basis for this pattern in that a collection of services that conform to one model establish a logical architectural layer of related functionality (Figure 6.21).

Applying Service Layers ensures that services matching common types are designed with the same fundamental characteristics, as derived from common service models. These services can form logical domains within the inventory, which can be evolved and governed as groups (Figure 6.22).

NOTE

Because service layers are closely related to service models, this pattern could have easily been called Service Models instead. The term “Service Layers” was chosen because it represents the end-result of repeatedly applying service models to service inventories.

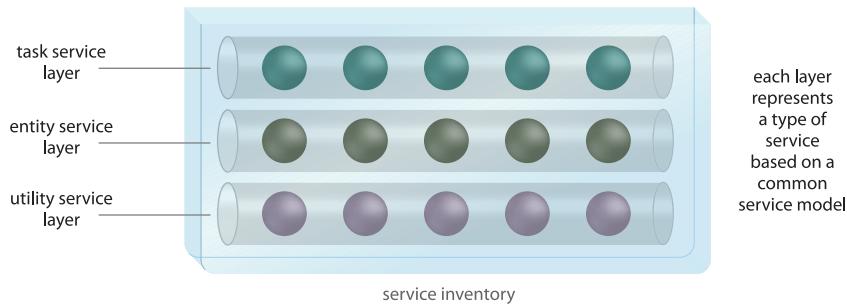


Figure 6.21

Related services are designed according to service models, thereby establishing logical service layers. In this case, the service inventory is structured with three service layers that correspond to the three abstraction patterns described in Chapter 7. (Note the pipe symbol is used to represent a service layer in this book.)

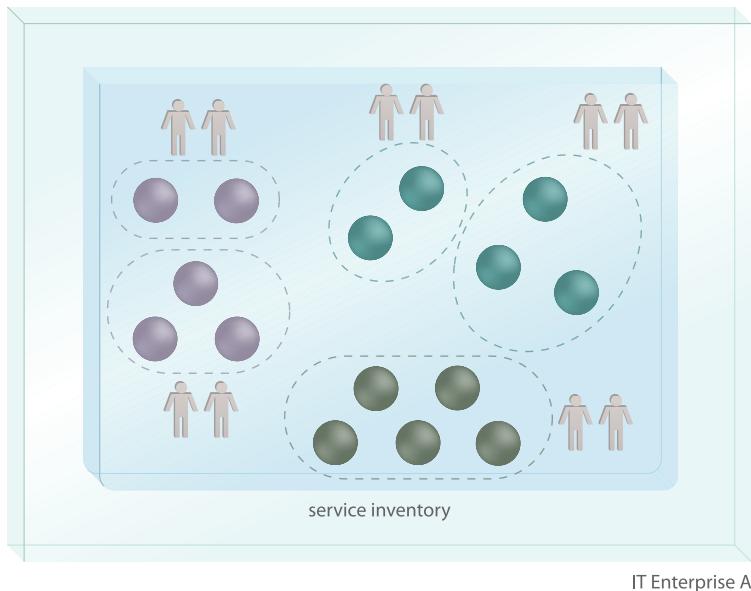


Figure 6.22

Layers (and sub-layers) can form groups of services. Long-term governance ownership of these groups can be assigned to dedicated custodians most suited to the nature of the underlying service models.

Application

Layers of services are generally defined prior to the implementation of a service inventory. During the modeling of a service inventory blueprint, the functional nature of planned service candidates helps determine what layers are most suitable. Therefore, any given service inventory can have different layers. The only rule is that there be a minimum of two.

The most fundamental layers are:

- a layer that represents single-purpose (non-agnostic) logic
- a layer that represents multi-purpose (agnostic) logic

By abstracting non-agnostic logic into one part of an inventory, agnostic logic can be defined and evolved in support of fostering reusability.

While there is always the option of customizing service layers, the safest starting point is to base them on common industry service models. These fundamental models have been proven to solve known design problems and are further explored in Chapter 7 as part of the descriptions for Process Abstraction (182), Entity Abstraction (175), and Utility Abstraction (168). As illustrated in Figure 6.23, service compositions tend to naturally span service layers established by common service models.

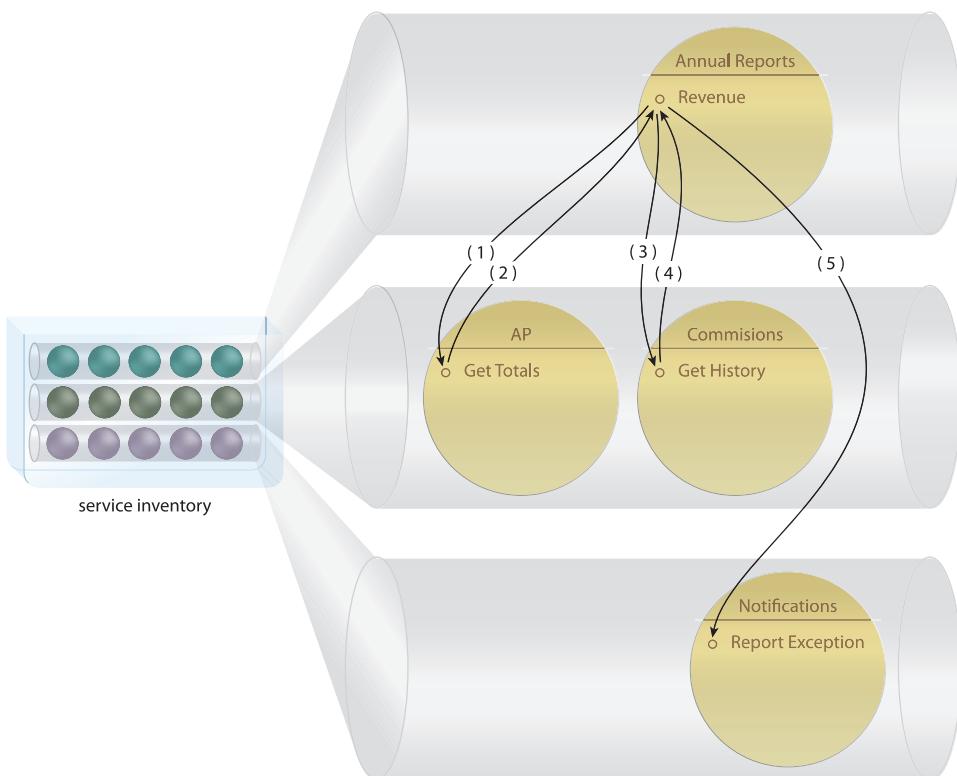


Figure 6.23

Shown here is a service inventory organized into process, entity, and utility layers (left) and a service composition drawn from this inventory comprised of services that span these layers (right).

Impacts

Determining what service models should or shouldn't be used within a service inventory requires a familiarity with the types of logic that reside within the inventory's boundary. Therefore, service layers often evolve out of repeated iterations of the service-oriented analysis phase and sometimes even require revisions to previously modeled service candidates. As a result, their use can increase the time and effort required to define a service inventory blueprint.

An exception to this is when service models have already been established as enterprise design standards, in which case they can be used as the basis for planned service layers right from the start.

Furthermore, once layers have been chosen, they become inventory-wide design standards in that every subsequently defined service needs to fit into one of the established service layers. After services have been built according to the underlying service models, it can be very difficult to change the structure of the established layers without disrupting the service inventory.

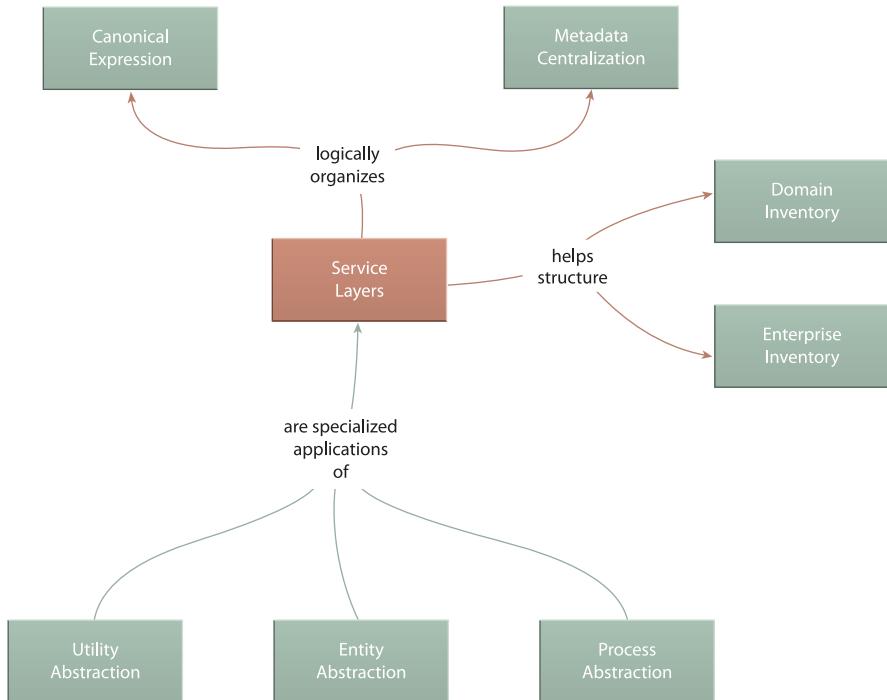
Relationships

Service Layers introduces logical separation into inventory boundaries and therefore naturally builds upon Enterprise Inventory (116) and Domain Inventory (123). Both Service Normalization (131) and Logic Centralization (136) help establish firm boundaries for individual services that allow for them to be organized into layers in support of this pattern.

As shown in Figure 6.24 and explored throughout Chapter 7, this pattern forms the basis of Utility Abstraction (168), Entity Abstraction (175), and Process Abstraction (182).

NOTE

Service Layers is at the root of the abstraction patterns that are core to the compound pattern Three-Layer Inventory (715).

**Figure 6.24**

Service Layers relates to preceding and upcoming patterns by adding a logical structure to the service inventory.

CASE STUDY EXAMPLE

As they proceed through service-oriented analysis stages, architects and analysts for Cutit, Alleywood, and the FRC individually decide that the task, utility, and entity service models will be used to standardize layers of services within their respective planned service inventories.

See the case study examples in Chapter 7 for details as to how specific layers are created by these organizations as a result of adopting the pattern.

6.3 Inventory Standardization Patterns

Design patterns and design standards were defined as two separate but related parts of a typical design framework back in Chapter 4. A design pattern provides a proven solution to a common design problem, and a design standard is a mandatory convention applied across multiple systems. Whereas a design pattern is industry-recognized, a design standard is internal and specific to an IT enterprise.

Even though they are distinct, design standards are a lot like design patterns. In fact, design standards can be seen as “pre-solving” specific design issues in order to ensure consistent system designs. It is therefore not uncommon for a design pattern to become the basis of a design standard, which is essentially what this group of patterns is all about.

These next patterns (Figure 6.25) do not just propose possible solutions to common problems, but they propose that to solve this set of specific problems, the solutions themselves must become actual design standards.

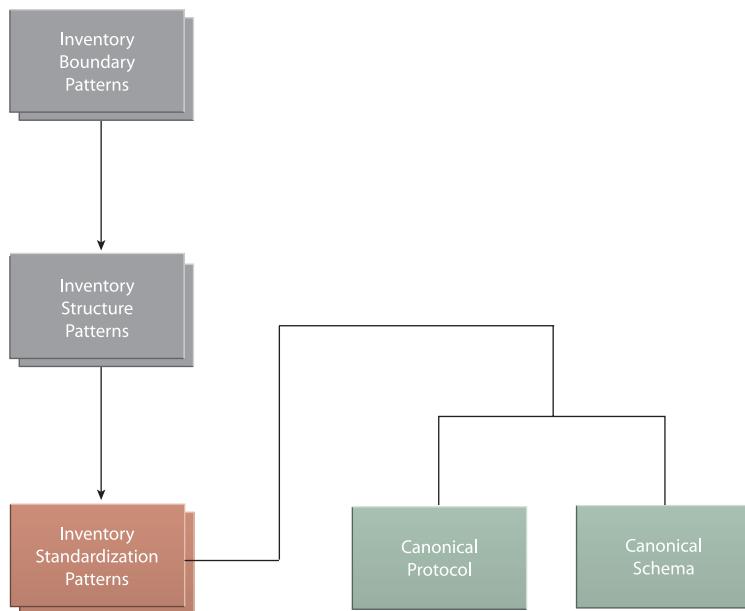


Figure 6.25

The two standardization patterns enforce fundamental design conventions in support of fostering intra-inventory interoperability.

Though there is no particular order in which these patterns need to be applied, it is sometimes necessary for Canonical Protocol to be established prior to the Canonical Schema because the communications technology represents the fundamental medium by which data (and associated data models) are delivered and processed.

Canonical Protocol

How can services be designed to avoid protocol bridging?



Problem	Services that support different communication technologies compromise interoperability, limit the quantity of potential consumers, and introduce the need for undesirable protocol bridging measures.
Solution	The architecture establishes a single communications technology as the sole or primary medium by which services can interact.
Application	The communication protocols (including protocol versions) used within a service inventory boundary are standardized for all services.
Impacts	An inventory architecture in which communication protocols are standardized is subject to any limitations imposed by the communications technology.
Principles	Standardized Service Contract
Architecture	Inventory, Service

Table 6.7

Profile summary for the Canonical Protocol pattern.

What Do We Mean by “Protocol”?

Within the context of this pattern, “protocol” represents technologies required to establish baseline communication. From a Web services perspective this would include the standardization of a transport protocol, such as HTTP, as well as a messaging protocol, such as SOAP. Other more specific message formats that may also be commonly referred to as protocols, would likely not fall within the scope of this pattern, especially if they introduce specific structures via pre-defined schemas. In this case, they may be more relevant to Canonical Schema (158).

Problem

Each service exists as a standalone software program. When these programs need to exchange information, they have to form a connection using a communications technology. When programs are designed to use different communications technologies, they are incompatible and cannot exchange information without involving a separate program that can translate one communications technology to another, as per Protocol Bridging (687).

Building services with different implementation technologies is not uncommon, but allowing services based on different communication technologies to exist within the same architecture can result in limitations. For example, groups of services based on the same communication framework are likely to be delivered as part of the same project. The day any of these services needs to be pulled into a new composition consisting of services delivered by a different project (and using different communication protocols), incompatibility issues could make their connectivity and reuse challenging and perhaps impossible (Figure 6.26).

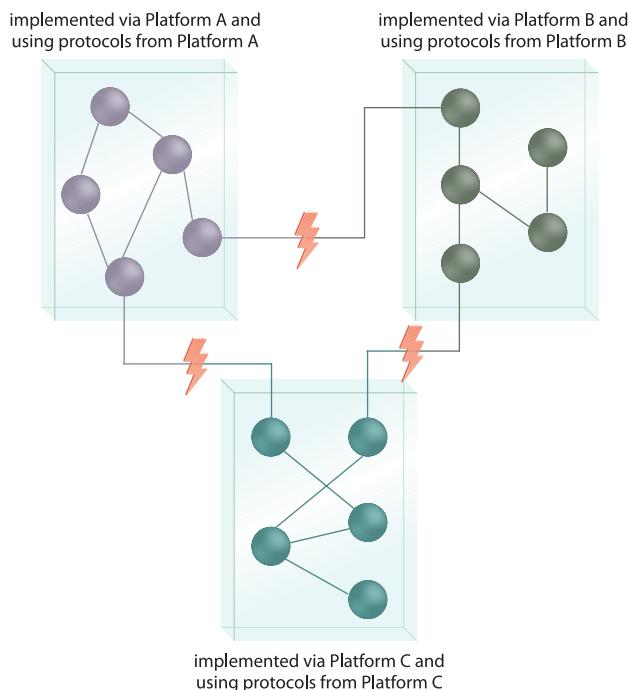


Figure 6.26

Different groups of services (likely delivered via different projects) establish communication boundaries through the use of incompatible communication protocols.

Solution

The technology architecture is designed to limit enablement of cross-service interaction to a single or primary communications protocol or protocol version. All other technologies associated with supporting the protocol's underlying communications framework are also standardized. This guarantees baseline technological compatibility across services (Figure 6.27).

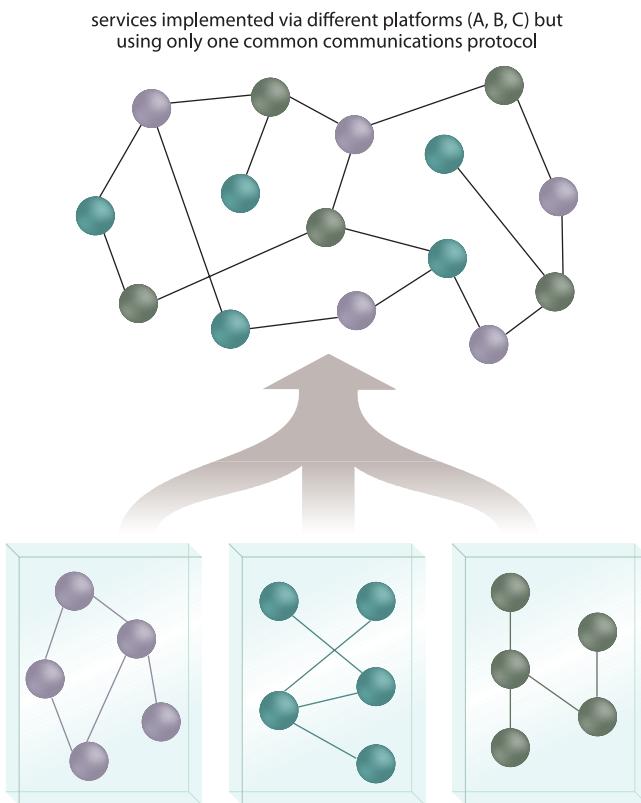


Figure 6.27

Though still delivered by different projects via different vendor platforms, these services conform to one centralized communications technology, making them technologically compatible.

NOTE

This design pattern advocates the standardization of protocols used for inter-service communication only. Traditional protocols, such as those used to communicate with proprietary components or databases, are not affected by this pattern as long as they remain part of the logic encapsulated by services.

Application

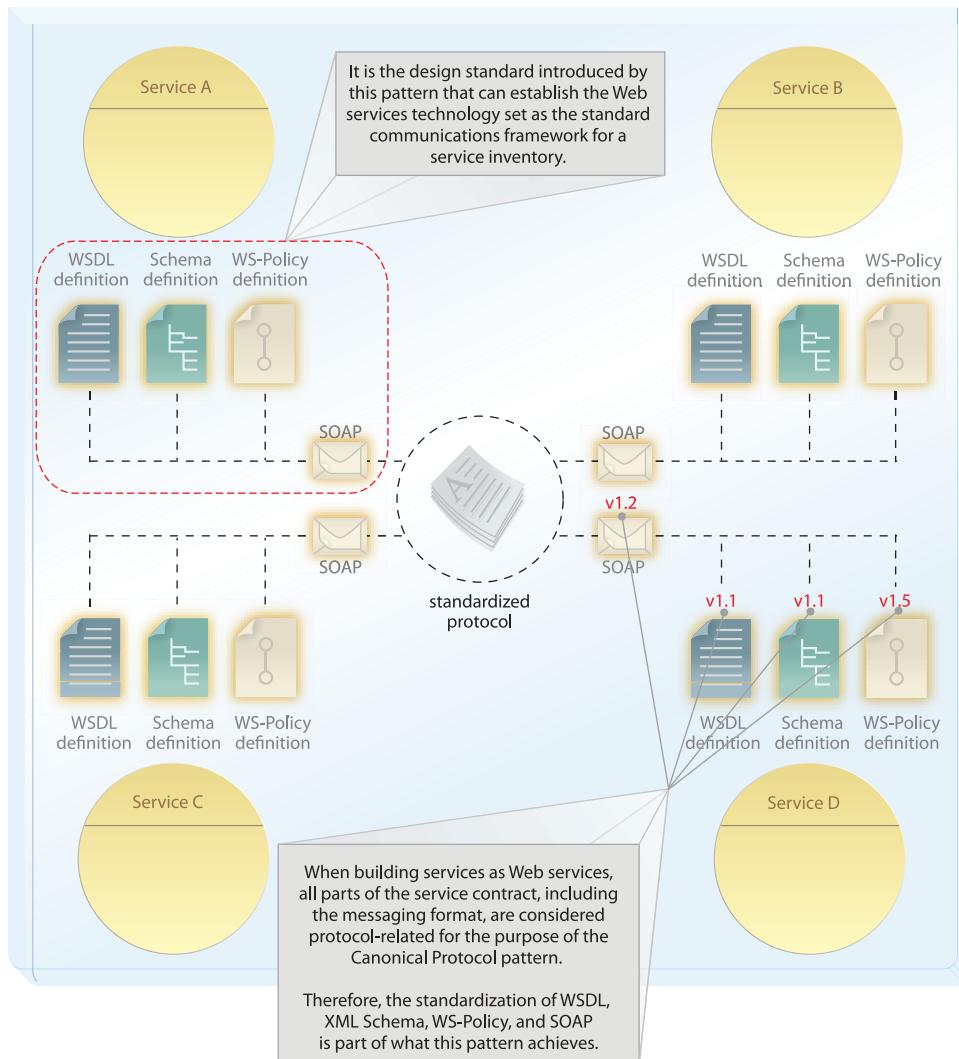
To ensure that all services an inventory architecture is intended to support can effectively interact and be repeatedly recomposed requires that a centralized communications technology be carefully chosen.

A common framework that fulfills this role is the Web services platform because it leverages industry-standard transport and messaging protocols (e.g., HTTP and SOAP) that are widely supported yet still vendor-neutral. However, even when using Web services, this design standardization pattern must still be applied to overcome possible disparity resulting from the mismatch of protocol-related *versions* (as illustrated in Figure 6.28). The WS-I Basic Profile is therefore likely a key part of applying this pattern as a means of ensuring technological compatibility among the various versions of Web service technology standards.

Alternative communication options can also be explored within controlled environments. For example, a proprietary vendor protocol can be chosen, as long as all services within its inventory are standardized to conform to its use.

NOTE

When applying this design pattern to Web services, any Web services-related industry standards and technologies associated with inter-service communication are affected. This can include WSDL, XML Schema, SOAP, WS-Policy, and various WS-* standards. This pattern does not dictate how these technologies are applied, only that their use—and, in particular, their version—are standardized.

**Figure 6.28**

All parts of the Web service contract are affected by this design pattern.

Impacts

Some key considerations when standardizing on one communication protocol are:

- *Maturity and Reliability* – Whichever protocol is chosen, service interaction throughout the resulting technology architecture will be constrained by whatever limitations this framework imposes. Therefore, the maturity and overall adequacy of the communications technology must be carefully assessed.
- *Longevity* – If there are any concerns that vendors may discontinue or abandon the technology, the associated risks would need to be taken into account.
- *Cost* – Building services to support a primary communication protocol can bring with it a series of hidden expenses. Some may be related to accommodating deficiencies in the protocol (as per the maturity, reliability, and longevity considerations just raised), while other costs can be incurred if the protocol is part of a proprietary platform that requires licensing fees.

When the preferred protocol imposes constraints in any of these areas, Dual Protocols (227) can be viewed as a viable alternative to (or even a first step toward) this pattern.

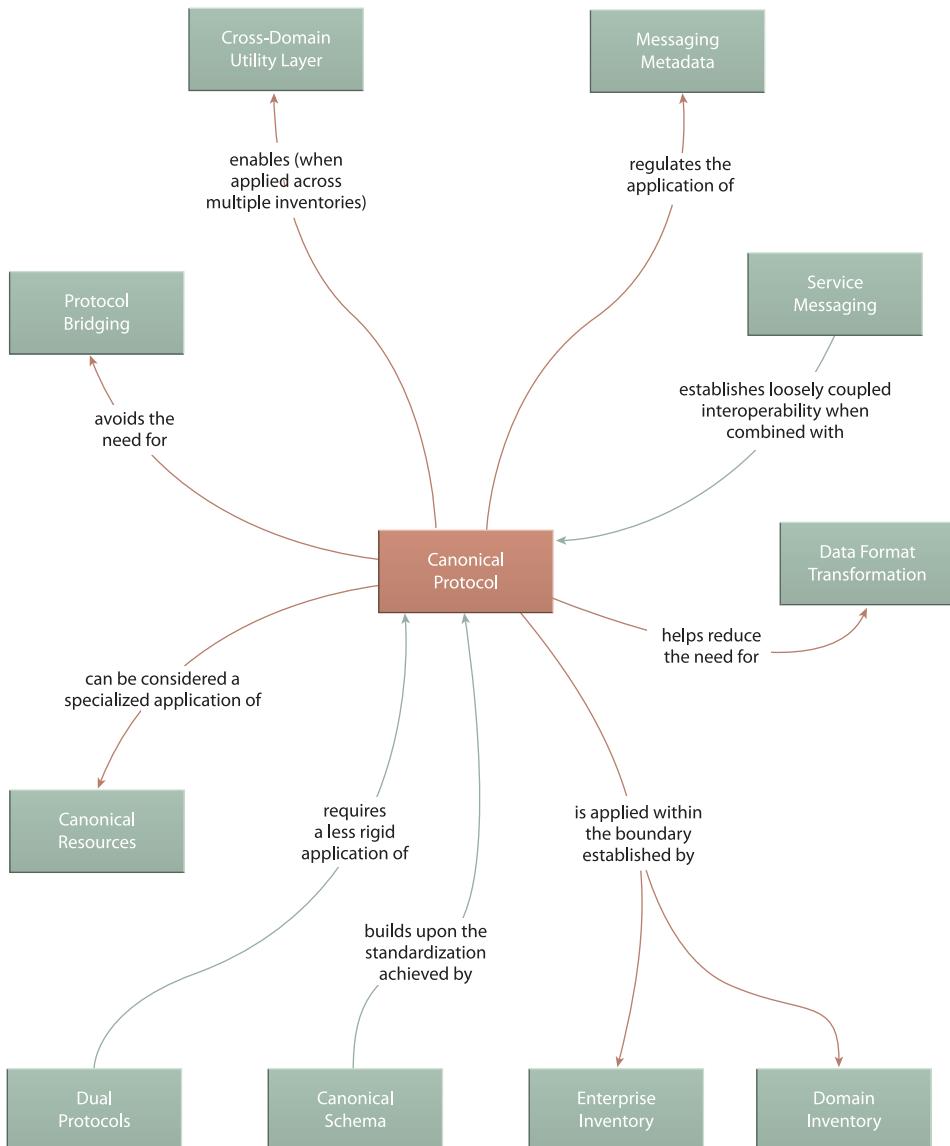
NOTE

This design pattern could be considered a specialized implementation of Canonical Resources (237) in that it is very much about standardizing technology across an inventory. However, it is singled out here because a communication protocol is more than just an architectural extension. It represents the fundamental means by which all parts of a distributed solution work together.

Relationships

Canonical Protocol's architectural focus naturally results in relationships with other architecture-centric design patterns, such as Canonical Resources (237), Enterprise Inventory (116), and Domain Inventory (123).

By standardizing the medium by which services exchange business and activity data, a foundation for service interoperability is established. Therefore, this pattern is closely related to Service Messaging (533), Messaging Metadata (538), and also Schema Centralization (200).

**Figure 6.29**

Canonical Protocol proposes an inventory-wide design standard that solves foundational interoperability issues and therefore relates to inventory and messaging patterns.

CASE STUDY EXAMPLE

Alleywood's IT environment has historically been Java-based. The services planned as part of the current project were naturally going to be developed and implemented using Java technologies. The importance of Canonical Protocol was acknowledged as architects determined that it would make the most sense for all services to be accessible via a central JMS messaging framework.

However, subsequent discussions with Tri-Fold architects and enterprise architects from McPherson resulted in a requirement to move toward a Web services-based communications framework instead.

This was a strategic decision based primarily on the fact that Tri-Fold's services are comprised of endpoints into a larger ERP, plus a set of custom .NET components. Neither environment supports JMS, but both support Web services, as does Alleywood's Java-based platform (Figure 6.30).

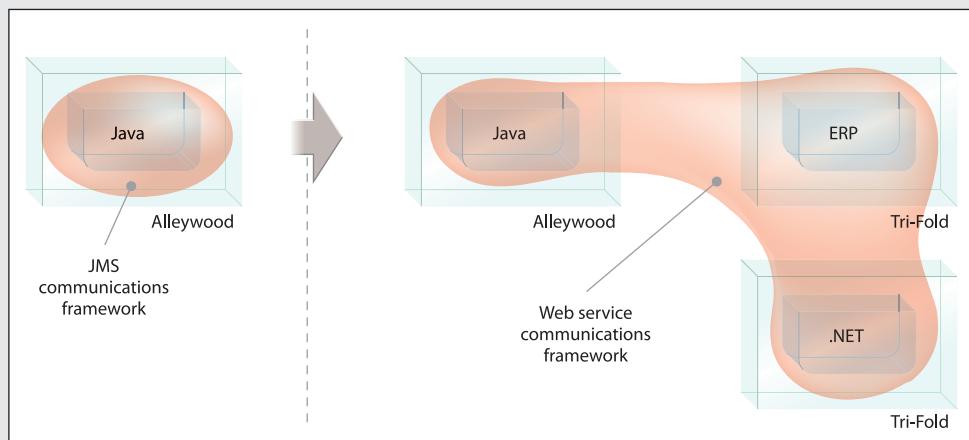
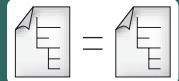


Figure 6.30

Although the application of this pattern was already considered when attempting to standardize on JMS, subsequent requirements broadened the scope of the centralized communications framework to encompass three platforms. This led to the decision to standardize on the use of Web services.

Canonical Schema

How can services be designed to avoid data model transformation?



Problem	Services with disparate models for similar data impose transformation requirements that increase development effort, design complexity, and runtime performance overhead.
Solution	Data models for common information sets are standardized across service contracts within an inventory boundary.
Application	Design standards are applied to schemas used by service contracts as part of a formal design process.
Impacts	Maintaining the standardization of contract schemas can introduce significant governance effort and cultural challenges.
Principles	Standardized Service Contract
Architecture	Inventory, Service

Table 6.8

Profile summary for the Canonical Schema pattern.

NOTE

Canonical Schema should not be confused with Canonical Data Model (Hohpe, Woolf). With Canonical Schema, services and consumers utilize and conform to an already-developed data model, avoiding the need for transformation. The classic Canonical Data Model pattern assumes that data model transformations are necessary and recommends that they be designed in such a manner that they adhere to a standard data model instead of resulting in pair-wise permutations.

Problem

For a service to send or receive data, it needs to know in advance exactly how that data will be organized and structured. For example, a business document such as an invoice can have its own data model structure that determines how invoice information is organized, what the different parts of an invoice document are called, and what data types and validation constraints should be associated with these parts.

When different services are delivered by different project teams, each team may decide to structure an invoice data model in a different way. When those services need to exchange invoice data at a later point in time, they will not be compatible and will require Data Model Transformation (671) to convert one invoice document structure into another.

This generally introduces the need to design and develop a custom transformation layer consisting of mapping logic and rules that resolve differences between disparate schemas. This logic can be implemented as part of the service hosting environment and carries out its transformation *every time* data needs to be exchanged via the affected service capabilities (and is therefore considered undesirable).

Solution

The need for Data Model Transformation (671) can be avoided by ensuring that service contracts are designed with compatible schemas from the beginning. This is achieved by applying data design standards to the data models within service contracts (see Figure 6.31).

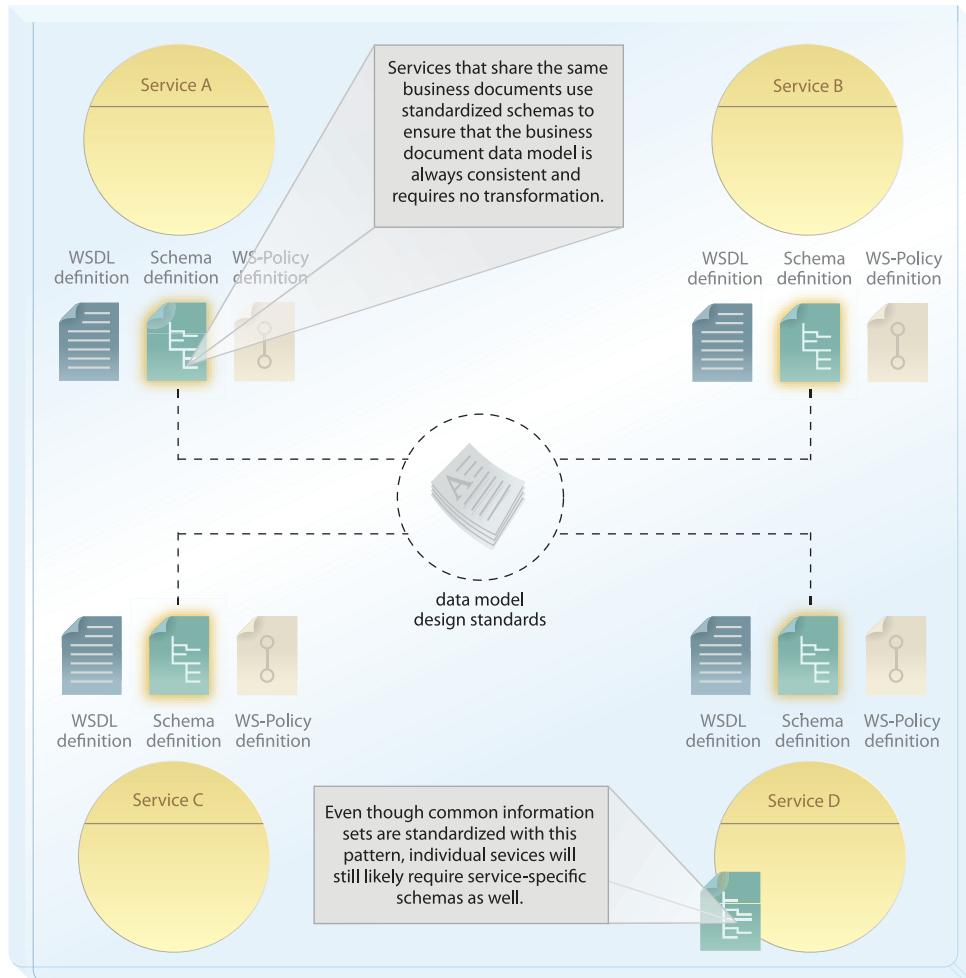
Application

Canonical Schema is commonly applied to services implemented as Web services because this allows for data models to be defined using the industry standard XML Schema expression language. In this case, XML Schema definitions representing the same type of documents or information sets need to be kept in alignment so that complex and simple data types remain in sync across different service contracts.

Once standardized schemas are in place, this pattern is realized via a formal process through which service contracts are designed, which ensures consistent application of the Standardized Service Contract design principle.

Impacts

In larger enterprises, the scope of data model standardization may need to be limited to individual domains so as to make the standardization effort and the subsequent governance responsibilities more manageable. In fact, it is the considerations raised by this pattern that often motivates organizations to apply Domain Inventory (123) over Enterprise Inventory (116).

**Figure 6.31**

Multiple services implemented as Web services have standardized XML schema definitions as a result of applying this pattern.

Relationships

Data models are typically standardized within the boundaries of inventories defined by Enterprise Inventory (116) or Domain Inventory (123), resulting in a healthy inventory architecture with a reduced need for Data Model Transformation (671).

Also, by establishing a key design standard in support of service interoperability, Canonical Schema forms several close relationships with other patterns, such as Schema Centralization (200) and Canonical Protocol (150).

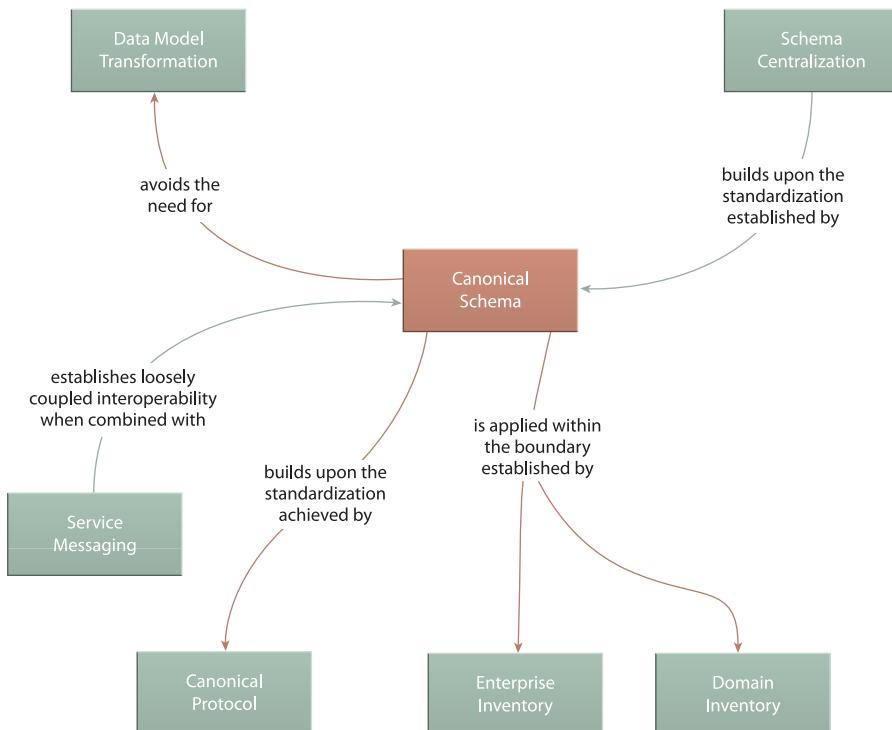


Figure 6.32

Canonical Schema achieves an important type of standardization that is fundamental to inventory architectures.

CASE STUDY EXAMPLE

The three disparate environments that were somewhat unified by the application of Canonical Protocol (150) and the introduction of a Web services framework are expected to share a fair amount of data.

Architects realize that the mere use of Web services will not standardize data exchange beyond the communication protocol itself. As it currently stands, the same types of data used by the individual environments is represented by different data models, as defined by the underlying database structures that were independently created. For an order

record to be moved from a Tri-Fold system to an Alleywood system, for example, a broker needs to be involved in order to transform one data model into another.

The fact that this issue brings with it many of the problems and challenges the project teams faced during past integration efforts gives them serious doubts as to the viability of their new Web services-based architecture. There is a realization that a further level of standardization is required.

Steps are taken to define “official” XML schemas to represent key business documents that will need to be shared across these systems. Web service contracts created as part of this framework are required to incorporate these schemas whenever they need to process data related to one of these business document types.

For the order record, a single Order schema is defined, and it is agreed that all order data passed between these environments will comply to the document structure and validation rules established by this schema (Figure 6.33).

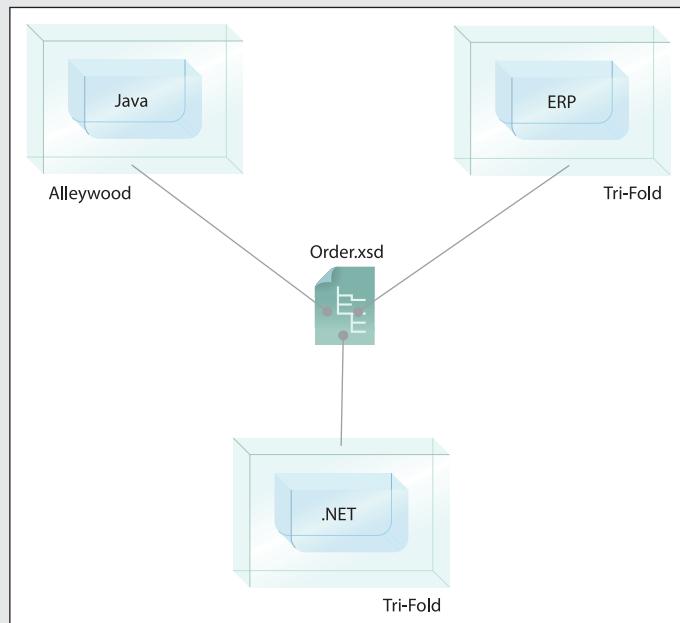


Figure 6.33

The application of this pattern ensures that a standardized schema is established for each key business document, such as one representing order information.

Chapter 7



Logical Inventory Layer Patterns

Utility Abstraction

Entity Abstraction

Process Abstraction

Service Layers (143) establishes a general means of organizing the services within an inventory into logical groups. Each layer is based on a type of service and therefore represents a set of services that conform to this type. These types correspond to industry classifications referred to as *service models*.

Following are the three most common service models:

- *Utility Service Model* – A type of service that provides generic processing logic that is not classified as business logic (as explained in the upcoming *Business Logic and Agnostic Logic* section). Utility logic is often referred to as “cross-cutting” logic because it is ideally agnostic and reusable and therefore multi-purpose in nature.
- *Entity Service Model* – A business-centric service type that is derived from one or more business entities. Entity services are also agnostic and therefore expected to be highly reusable.
- *Task Service Model* – Also a business service model, but one that is intentionally non-agnostic because its functional scope is limited to single-purpose business process logic.

These three service models correspond to the three inventory layer patterns described in this chapter as follows:

- Utility Abstraction (168) establishes a service layer comprised of utility services.
- Entity Abstraction (175) results in a service layer that represents entity services.
- Process Abstraction (182) creates a non-agnostic service layer that consists of task services.

Combining Layers

As explained by Three-Layer Inventory (715), it is a recommended practice to use all three of these design patterns together. For most organizations, they collectively represent an effective grouping of common service logic.

However, this is not absolutely required. The rule established by Service Layers (143) is that at least two layers must exist. Given that it is generally desirable to have one layer abstract non-agnostic logic, this means that Process Abstraction (182) can be applied together with either Utility Abstraction (168) or Entity Abstraction (175), as shown in Figure 7.1.

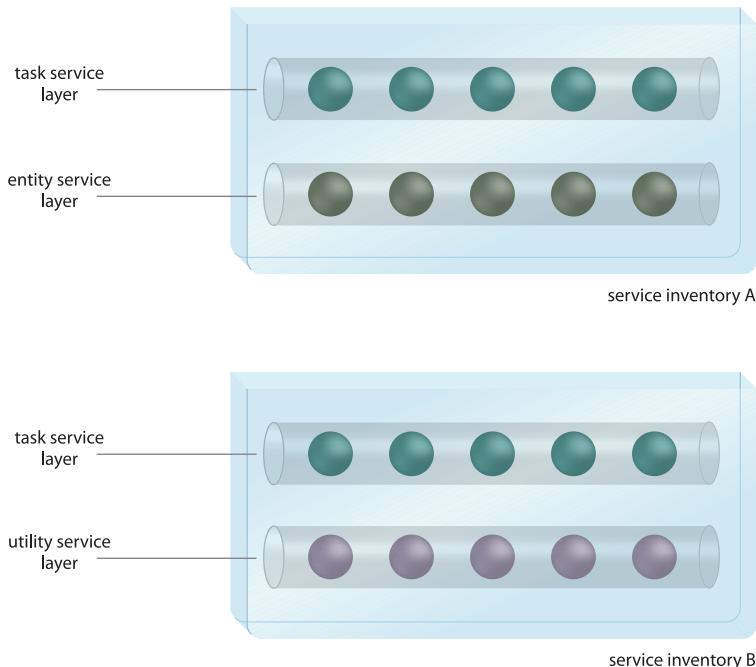


Figure 7.1

Two different service inventories, each with only two service layers. Inventory A establishes an entity service layer that is most likely comprised of services that also encapsulate common utility logic. Inventory B limits its agnostic services to the utility layer, which probably results in entity-specific logic being redundantly dispersed throughout the task service layer.

Service layer combinations that are less common include the following:

- *Utility + Entity* – If these layers are intended to represent agnostic and reusable services, then this layer combination will rely on composition via consumer programs that are not service-oriented. This can still foster reuse but the overall composability potential of services will be limited.
- *Task Only* – A service inventory structured with a single task service layer makes little sense because it would essentially be comprised of a series of independent, silo-based applications. Although it is common to find a set of single-purpose programs that represent some or all of an IT enterprise, from a service-orientation perspective, there is not much gain to this design. Agnostic service layers are required to establish services as reusable enterprise resources.

NOTE

Organizations can choose to derive custom variations of these fundamental abstraction patterns and can even create new service models and abstraction layers altogether. Sometimes this approach is warranted when a service inventory spans domains and a unique, business domain-specific service model is required.

Business Logic and Utility Logic

When discussing service models and service layers (and service design in general), a distinction is always made between business logic and non-business logic. Logic is classified as being business-centric when it is derived from business analysis models and specifications. Examples of such documents include workflow or business process definitions, BPM specifications, ontologies, taxonomies, logical data models, business entity references diagrams, and a variety of other documents related to business architecture, data architecture, and information architecture in general.

Anything having to do with *representing* the manner in which an organization carries out its business can generally be classified as a form of business logic. When it comes to service encapsulation, we are primarily interested in business logic that can be automated.

Automating business logic requires more processing than is generally documented by the aforementioned business analysis documents. There are various underlying mechanics and resources that come into play at a technology level. Those parts of the processing logic that are not related to or derived from business logic are classified as utility logic, as explained in the upcoming description of Utility Abstraction (168).

Agnostic Logic and Non-Agnostic Logic

The term “agnostic” originated from Greek where it means “without knowledge.” Therefore, logic that is sufficiently generic so that it is not specific to (has no knowledge of) a particular parent task is classified as *agnostic* logic. Because knowledge specific to single purpose tasks is intentionally omitted, agnostic logic is considered multi-purpose. On the flipside, logic that *is* specific to (contains knowledge of) a single-purpose task is labeled as *non-agnostic* logic.

Another way of thinking about agnostic and non-agnostic logic is to focus on the extent to which the logic can be repurposed. Because agnostic logic is expected to be multi-purpose, it is subject to the Service Reusability principle with the intention of turning it into highly

reusable logic. Once reusable, this logic is truly multi-purpose in that it, as a software program (or service), can be used to automate multiple business processes.

Non-agnostic logic does not have these types of expectations, which is why non-agnostic services are deliberately designed as single-purpose software programs.

NOTE

The word “agnostic” also has specific meaning within some religious communities. If you are uncomfortable using this term, you can substitute it with terms like “neutral” or “unbiased.” Although the underlying meaning is not quite as clear with these terms, they may still be effective in making the distinction between these logic types.

Service Layers and Logic Types

Each of the design patterns in this chapter defines a service layer that is based on a distinct combination of the four logic types we just covered, as shown in Table 7.1.

	Business Logic	Utility Logic	Agnostic Logic	Non-Agnostic Logic
Utility Service Layer		x	x	
Entity Service Layer	x		x	
Task Service Layer	x			x

Table 7.1

An overview of how common service layers relate to the fundamental logic types. Any service that ends up containing logic that spans two or more layers cannot be cleanly grouped into a layer structure such as this, and is therefore often labeled as a *hybrid service*.

Note the absence of a service layer that represents both utility and non-agnostic logic. Such a service layer can be created, but it is not common and therefore not documented as a separate pattern. Generally, the need to assemble services together into compositions is driven by business-centric tasks or processes. This establishes the task service layer as the primary part of a service inventory that abstracts non-agnostic logic.

Utility Abstraction

How can common non-business centric logic be separated, reused, and independently governed?



Problem	When non-business centric processing logic is packaged together with business-specific logic, it results in the redundant implementation of common utility functions across different services.
Solution	A service layer dedicated to utility processing is established, providing reusable utility services for use by other services in the inventory.
Application	The utility service model is incorporated into analysis and design processes in support of utility logic abstraction, and further steps are taken to define balanced service contexts.
Impacts	When utility logic is distributed across multiple services it can increase the size, complexity, and performance demands of compositions.
Principles	Service Loose Coupling, Service Abstraction, Service Reusability, Service Composability
Architecture	Inventory, Composition, Service

Table 7.2

Profile summary for the Utility Abstraction pattern.

Problem

Among the logic required to automate just about any business task, there will be some that can be considered generic, “cross-cutting” processing functionality that has no relationship to formal business models. IT environments typically have a variety of technologies, products, databases, and other resources that offer features or functions useful for many purposes. This type of non-business centric logic can be considered utility logic.

The functionality associated with the automation of a business process will often include utility processing functions that find themselves bundled together into the same service with business process logic, business rules, and other forms of business logic (Figure 7.2).

This packaging results in hybrid services that make the individual strategic design and governance of utility logic practically impossible. For example, if generic processing

functionality capable of addressing multiple cross-cutting concerns is embedded together with business process-specific logic, it becomes challenging to make the generic processing logic separately available for reuse.

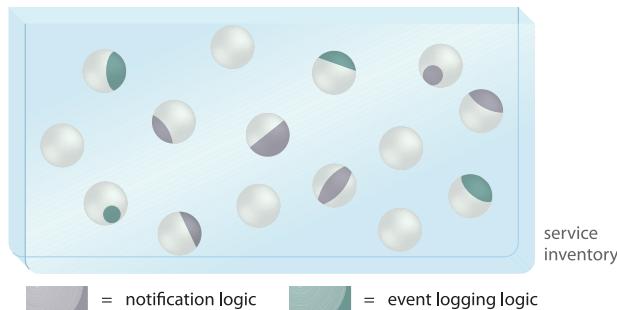


Figure 7.2

Utility logic is embedded within services that also contain business-centric functionality. As a result, much of the utility logic is redundantly implemented and not reusable.

Solution

Agnostic non-business-centric utility functions are defined and grouped into separate utility services. Because these utility services provide common functions that are not specific to any one task, they can be reused to automate multiple tasks. The result is a utility service layer (Figure 7.3) that is typically defined, owned, and governed by technology experts (usually without involvement of business experts).

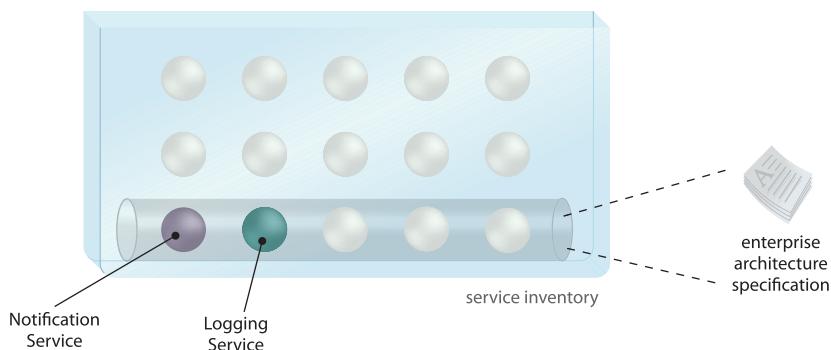


Figure 7.3

Cross-cutting utility logic is identified with the help of enterprise technology architecture specifications and then abstracted into a layer of dedicated services based on the utility service model.

Application

Utility processing is common to all enterprises, but the process of abstracting cross-cutting functionality into reusable units of logic can be difficult. One challenge constantly associated with utility service designs is the definition of appropriate service contexts. Unlike business service contexts that can be derived from existing business models, the functional context of utility services is often left to the judgment of architects and developers. It can therefore be challenging to set a service context that is suitable for long-term reuse and service contract longevity.

Here are some guidelines:

- Avoid overly coarse-grained services that bundle lots of capabilities together. These can be difficult to reuse and establish awkward functional contexts that can lead to bloated services over time.
- Define a very clear functional context for each service but give it the flexibility to evolve with the inventory. Unlike business services that tend to have strict boundaries, utility service boundaries can be augmented somewhat as long as the parent context is preserved.
- Use Canonical Expression (275) to ensure the creation of easy-to-understand service contracts. Because utility services tend to be produced by technology experts, there is often the danger that their public-facing contract details will be too technology-centric and cryptic.

During the service modeling process, the logical utility layer is already preconceived and conceptualized. Subsequently, when service contracts are ready to be defined, a special process geared toward utility service design needs to be applied so that the unique issues associated with this type of service can be addressed.

Establishing a formal utility service layer that spans a service inventory requires constant attention to how logic is partitioned and grouped within functional service contexts. Despite best efforts, you should be prepared to eventually split up coarse-grained utility services, as per Service Decomposition (489). You can prepare for this by applying Decomposed Capability (504) in advance.

Impacts

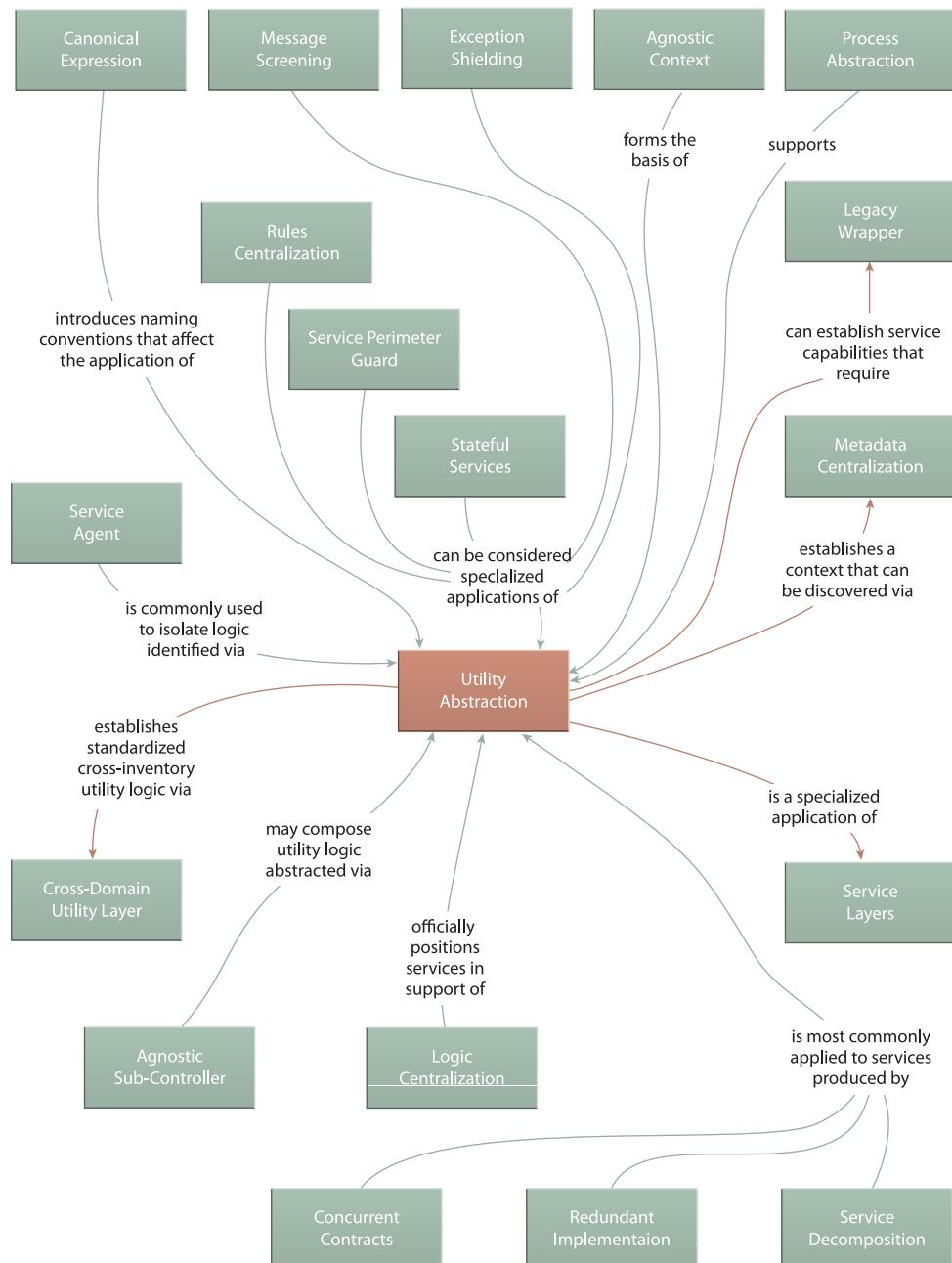
Adding this service layer to an inventory that already separates its services into multiple business-centric layers will predictably increase the size and design complexity of service compositions. Due to the additional inter-service communication required, runtime performance will also be affected.

Furthermore, the definition of utility service layers can make some impositions on how traditional development projects may have been carried out. However, the fact that object-oriented analysis and design (as well as aspect-oriented programming) have raised an awareness of the benefits of abstracting cross-cutting utility logic, these requirements will not be too foreign to most organizations.

Relationships

Because its application also results in agnostic service layers—and therefore is fundamentally influenced by Agnostic Context (312)—Utility Abstraction shares many of the same pattern relationships as Entity Abstraction. The primary difference is the absence of business-centric influences.

Notable relationships specific to Utility Abstraction are Service Agent (543), which emulates its non business-centric functional context and Cross-Domain Utility Layer (267), which essentially results in a broad application of Utility Abstraction. Rules Centralization (216), Service Perimeter Guard (394), and Stateful Services (248) also can be considered specialized implementations of this pattern.

**Figure 7.4**

Utility Abstraction tends to relate to design patterns that are not business-centric but still concerned with the design of agnostic logic.

CASE STUDY EXAMPLE

Cutit's immediate priority is to streamline their internal supply chain process. The order process in particular needs to be supported by the planned services so that orders and back-orders can be fulfilled as soon as possible.

A service-oriented analysis effort is carried out with the assumption that all three common service layers (utility, entity, task) will be used as the basis for the inventory's logical structure. This stage includes detailed service modeling and business process decomposition, resulting in the identification of several key relationships between different Cutit artifacts.

Here are some examples:

- Everything originates with the manufacturing of chain blades in the Cutit lab, which requires the use of specific *materials* that are applied as per predefined *formulas*.
- The assembly of *chains* results in products being added to the overall *inventory*.
- *Saws* and *kits* are items Cutit purchases from different manufacturers to complement their chain models.
- *Notifications* need to be issued when stock levels fall below certain thresholds or if other urgent conditions occur.
- Finally, a periodic *patent sweep* is conducted to search for recently issued patents with similarities to Cutit's planned chain designs.

These artifact relationships are incorporated into service modeling steps that produce the preliminary service inventory blueprint shown in simplified form in Figure 7.5. The analysis reveals a series of service candidates, most of which are business-centric in nature. The Patent Sweep and Notifications service candidates, however, have functional contexts that do not correspond to any modeled business specifications. They are therefore classified as utility services and together establish the beginning of Cutit's inventory utility service layer.

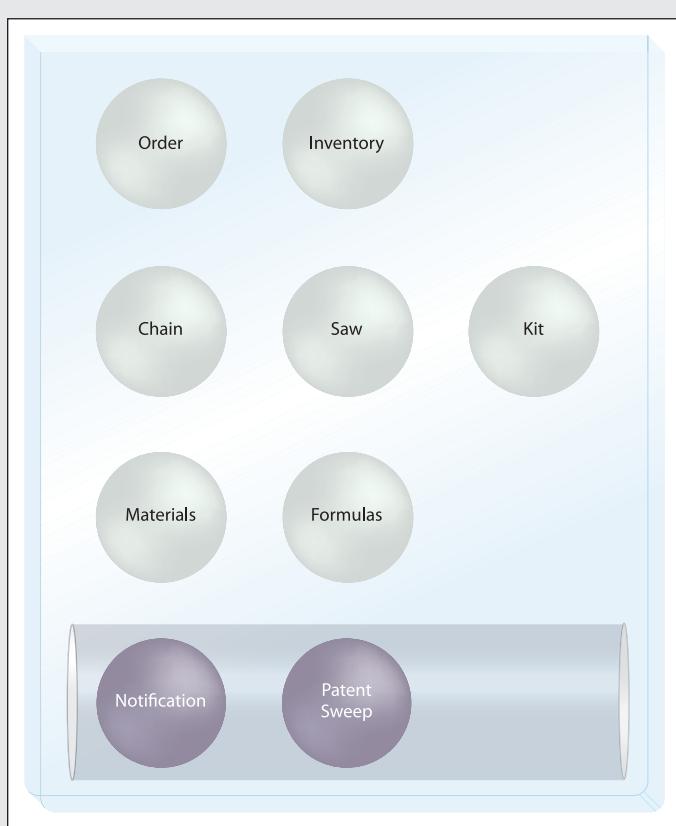


Figure 7.5

The initial set of services planned to support the following types of processes: keeping track of orders and back-orders, chain manufacturing, tracking required manufacturing materials, and the inventory management of manufactured and purchased products. To support these tasks, two utility services are defined. Note that task services are intentionally excluded from this diagram so that they can be introduced in the case study example for Process Abstraction (182).

Entity Abstraction

How can agnostic business logic be separated, reused, and governed independently?



Problem	Bundling both process-agnostic and process-specific business logic into the same service eventually results in the creation of redundant agnostic business logic across multiple services.
Solution	An agnostic business service layer can be established, dedicated to services that base their functional context on existing business entities.
Application	Entity service contexts are derived from business entity models and then establish a logical layer that is modeled during the analysis phase.
Impacts	The core, business-centric nature of the services introduced by this pattern require extra modeling and design attention and their governance requirements can impose dramatic organizational changes.
Principles	Service Loose Coupling, Service Abstraction, Service Reusability, Service Composability
Architecture	Inventory, Composition, Service

Table 7.3

Profile summary for the Entity Abstraction pattern.

Problem

When attempting to abstract business logic there is a natural tendency to group together logic associated with a specific task or business process. Any potentially reusable business logic is embedded together with single-purpose, process-specific logic. Therefore, the reusability potential for this logic is lost (Figure 7.6).

Additionally, the business analysts who have entity-level expertise are often different from those who have process-level expertise. When entity and process logic are grouped together in support of automating a particular task, it is usually owned by the analysts responsible for the business process definition. This can result in missed opportunities to incorporate design considerations specific to business entity rules, characteristics, and relationships.

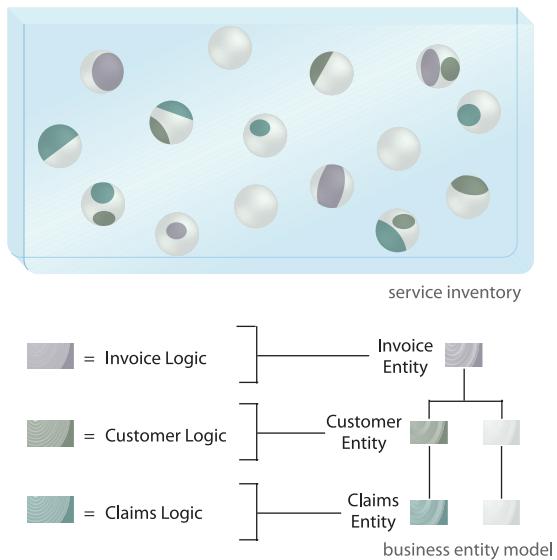


Figure 7.6

Solution logic associated with the processing of specific business entities is added to (most likely task-centric) services as required and therefore is dispersed (and redundantly implemented) throughout the service inventory.

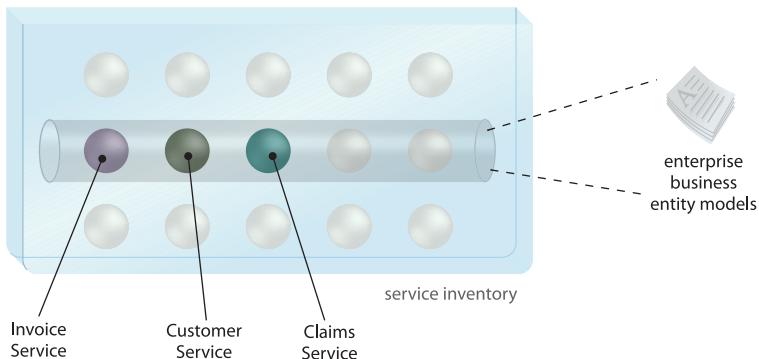
Solution

To carry out its business, each organization deals with different “business things,” like people, documents, products, and partner organizations. These things (or artifacts) are referred to as *business entities*. As organizations change the way they do business, new tasks may be required, or existing tasks may need to be altered. But throughout all of this change, new or revised tasks usually continue to involve the same business entities.

When looking for ways to design multi-purpose services that have a lot of reuse potential, it is therefore considered a safe bet to build services based on business entities. These entity services are naturally multi-purpose because each can be reused to help automate different tasks. This pattern partitions business logic that is evidently multi-purpose into a separate set of services with agnostic functional contexts based on business entities (Figure 7.7).

Application

To apply this pattern, the service modeling process needs to be carried out to identify and group logic appropriate for entity service encapsulation. Subsequently, the service-oriented design process for entity services must be completed to create standardized

**Figure 7.7**

A layer of entity services, each of which encapsulates processing associated with a specific business entity (or a group of related entities).

service contracts based on business entity contexts. Often a logical data model or an enterprise entity model provides the source for these contexts.

The resulting service layer is comprised of a (usually large) collection of agnostic business services that can be reused across numerous different business processes. Each entity service can be owned and governed by a group that includes business analysts with the appropriate subject matter expertise to preserve its integrity and to ensure the service continually evolves in alignment with the business.

Note that the granularity of entity services can sometimes vary. Although deriving a single service context from a single business entity results in a cleanly modeled service layer, this approach is not always possible. Practical considerations sometimes require that a service context be based on multiple entities—or a single entity may form the basis for multiple service contexts (Figure 7.8).

NOTE

This pattern may not be suitable for organizations with business entities that are volatile and subject to frequent change. In this case other, more stable sources for agnostic business contexts need to be sought. Alternatively, a new business entity specification can be developed wherein more abstract business entities are defined that are less prone to change. For example, in a company that has constantly changing products, it can be more effective to base entity service contexts on an abstract product entity instead of individual product types that may have limited longevity.

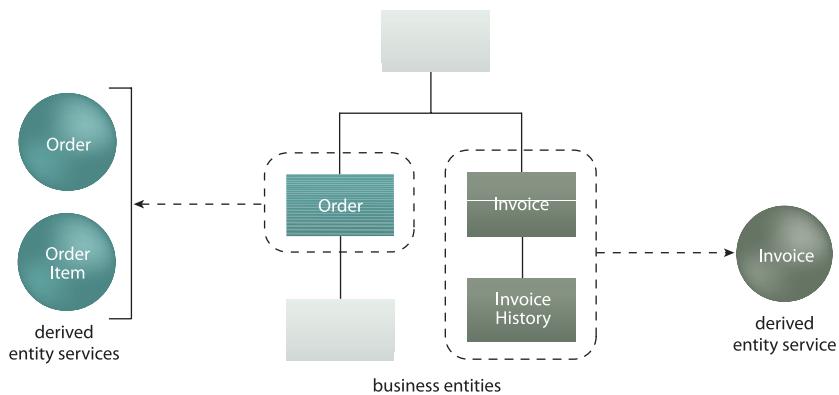


Figure 7.8

The parts of a business entity model encapsulated by entity services can vary.

Impacts

Although there is tremendous business benefit to establishing an entity service layer, it can impose change on several levels, not limited to just analysis and design processes. Because this pattern positions a significant portion of business logic as reusable enterprise resources (services), a great deal of attention needs to be focused on its subsequent governance and evolution.

The application of this pattern can shift organizational structures, change the complexion of project teams, and introduce new skill set requirements. Therefore, Entity Abstraction should be incorporated as early in the planning stages as possible, so as to give all of those involved with service modeling and service design enough time to understand and accept the nature of this service layer.

One of the key success factors to maintaining an effective entity service layer is establishing a suitable ownership structure for the entity services. Because this can necessitate joint ownership between business subject matter experts and technology experts, it may require the formation of new groups and policies.

Relationships

Entity Abstraction can be viewed as a business-centric application of Agnostic Context (312). It is therefore closely related to patterns that support the definition of agnostic business services, such as Logic Centralization (136).

As discussed in Chapter 16, coarse-grained entity services tend to require the application of Service Decomposition (489) at some stage. They also raise further business logic-related design considerations that carry over to contract design, which is where patterns such as Validation Abstraction (429) and Legacy Wrapper (441) may be required.

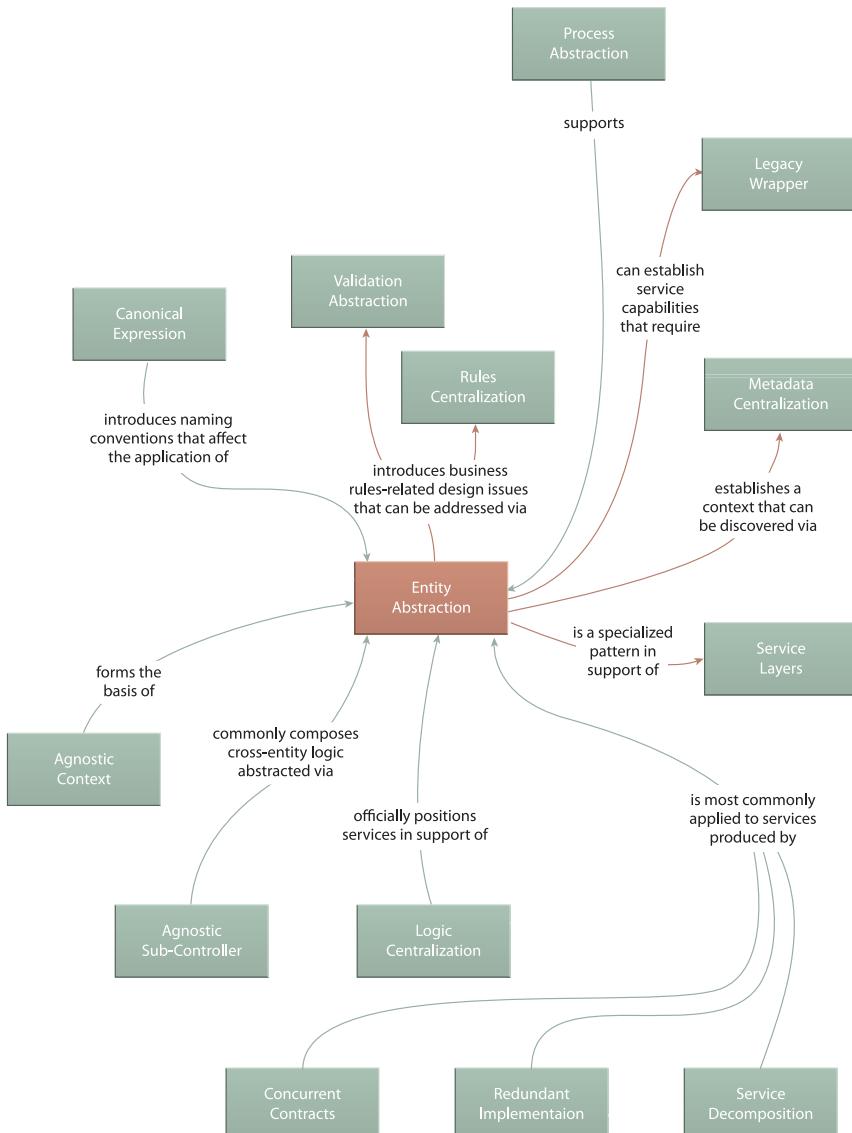


Figure 7.9

Entity Abstraction combines agnostic and business-centric functional contexts, which is why it relates to a range of different design patterns.

CASE STUDY EXAMPLE

Using an elaborate entity relationship diagram (that has long covered an entire wall in one of their IT meeting rooms), Alleywood architects collaborate with business analysts to carry out their own service-oriented analysis with a focus on identifying core business entities.

After iterating through and decomposing numerous business processes, a series of artifacts are documented, along with their inter-relationships:

- *Employees* use *equipment* to process natural wood in the field. The equipment needs to be maintained with an inventory of *parts*.
- Accumulated wood is placed on *trucks*, which haul *loads* to the *mills* where the wood is processed and refined.
- Mills are located in different *regions*, some of which are governed by different regulations.
- *Alerts* are issued for warning and emergency situations, such as when an employee is injured or a truck breaks down.
- *Policy checks* are periodically performed to look for amended or newly issued regulations that may affect existing wood processing plans.
- Finally, *resource surveys* are conducted to search through a central repository of workers available for field jobs.

Of these artifacts, it is determined that Alerts and Resource Survey are to be classified as utility service candidates because they do not correspond to recognized business entities.

There is some debate as to whether Policy Check should be considered an entity service even though their business entity diagram does not contain a policy entity. In the end, they decide to also classify this as a utility service candidate because they could not foresee the need for a functional context dedicated to policies (especially considering that the only policies they are concerned with are those issued by the FRC).

As shown in Figure 7.10, eight entity service candidates are defined, establishing a fair-sized entity service layer. With the exception of Employee, each of these entity-centric functional contexts corresponds to one business entity within their entity relationship diagram. The same business analysts responsible for maintaining that diagram agree to remain involved with the ownership and evolution of these entity services.

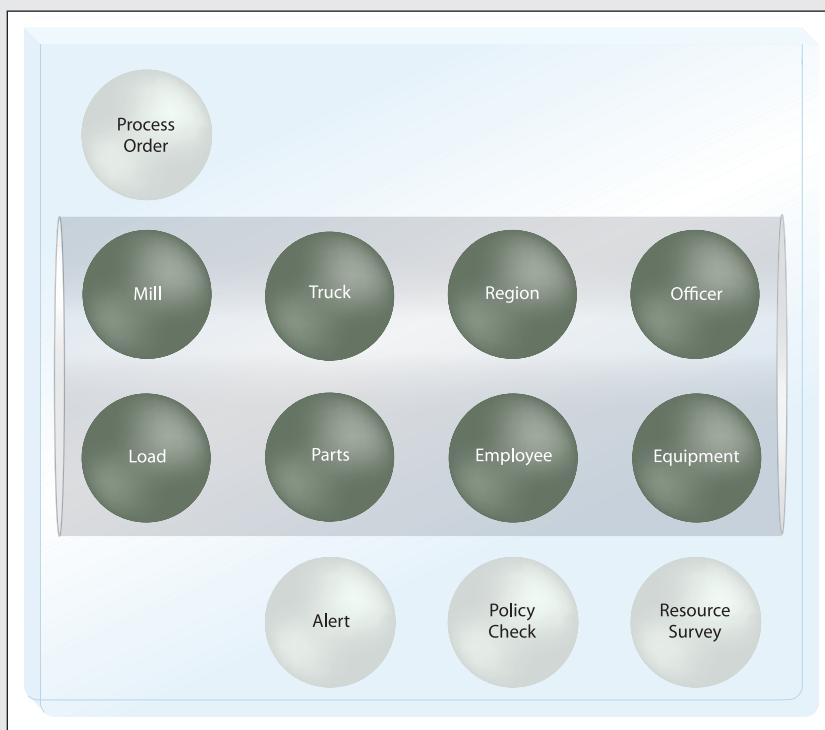


Figure 7.10

The services within the entity layer represent the bulk of the service logic defined so far and will therefore be required to shoulder a great deal of the processing associated with Alleywood's primary automation tasks.

The Employee service candidate was actually derived from three separate, employee-related business entities for which the creation of separate services was not deemed justifiable.

Note that because Alleywood does not have a single business entity to represent an order, a task service is established to encapsulate all of the composition logic required to process order documents.

Process Abstraction

How can non-agnostic process logic be separated and governed independently?



Problem	Grouping task-centric logic together with task-agnostic logic hinders the governance of the task-specific logic and the reuse of the agnostic logic.
Solution	A dedicated parent business process service layer is established to support governance independence and the positioning of task services as potential enterprise resources.
Application	Business process logic is typically filtered out after utility and entity services have been defined, allowing for the definition of task services that comprise this layer.
Impacts	In addition to the modeling and design considerations associated with creating task services, abstracting parent business process logic establishes an inherent dependency on carrying out that logic via the composition of other services.
Principles	Service Loose Coupling, Service Abstraction, Service Composability
Architecture	Inventory, Composition, Service

Table 7.4

Profile summary for the Process Abstraction pattern.

Problem

Services can be designed to resemble traditional silo-based applications wherein agnostic and non-agnostic logic is grouped together in each service. This can happen when services are delivered individually by separate project teams or service-orientation is disregarded as part of the delivery method.

This approach has several repercussions:

- It reduces opportunities for applying the Service Reusability design principle on a broad scale.
- It imposes governance complexity when expertise associated with business entities and business processes lie with different individuals.

- It makes it difficult to apply Non-Agnostic Context (319), thereby reducing the chances of successfully abstracting single-purpose cross-entity logic into legitimate services.

As illustrated in Figure 7.11, this grouping can further result in the fragmented implementation of task logic.

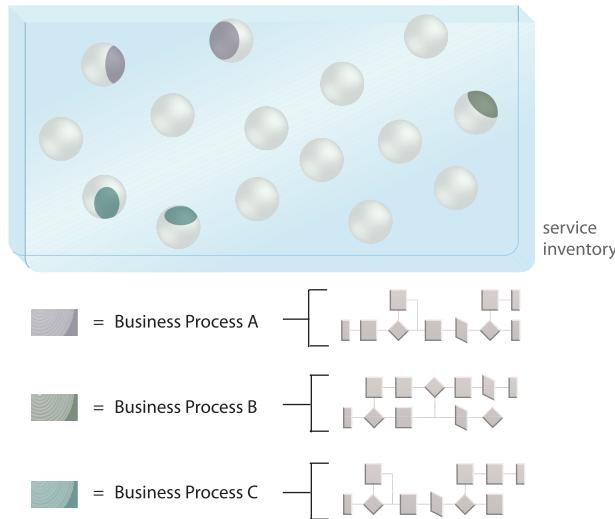


Figure 7.11

Parent business process-specific logic is grouped with other logic that is likely agnostic, resulting in some dispersal. The primary negative effect is that by combining task-specific and task-agnostic logic, the opportunity to establish agnostic services in support of Agnostic Context (319) and other related patterns, such as Logic Centralization (136), is hindered.

Solution

Business logic that spans multiple entity service boundaries is abstracted into a distinct functional context associated with the task service model. This establishes a parent service layer responsible for containing workflow and service composition logic required to carry out the parent business process (Figure 7.12).

The abstraction established by this process service layer can increase organizational agility because it is the parent business logic that is commonly subject to business change. As a result, being able to access and maintain this logic in a separate set of services can decrease

the effort required to respond to change while shielding agnostic services in other layers from the impact of the change. This is because when a mature inventory of services is available, business changes will often only translate into a need to recompose agnostic services without modifying them.

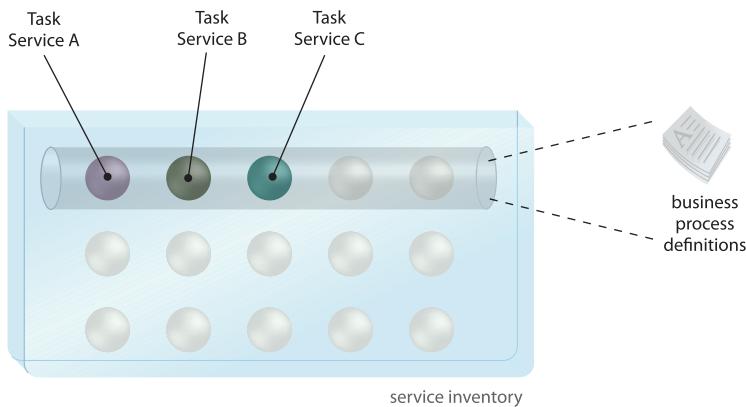


Figure 7.12

Solution logic limited to the fulfillment of parent business processes is abstracted into separate task services. This establishes a parent task service layer that abstracts non-agnostic business process logic responsible for composing agnostic services.

Application

It may appear as though this pattern is applied out of necessity in support of Utility Abstraction (168) and Entity Abstraction (175). Because these two patterns force the isolation of business process-agnostic logic, any logic that is specific to parent business processes must be located in its own layer.

However, logic residing in a parent business process layer does not need to be encapsulated by services. The formation of a task service layer is the result of repeatedly applying Service Encapsulation (305) and Non-Agnostic Context (319) to this logic so as to shape it into well-defined services.

Services based on a task-centric context are very similar in concept to traditional silo-based applications, in that they are associated with the execution of a specific business process. Therefore, these types of services are more easily incorporated into established project delivery lifecycles and subsequent ownership arrangements.

The intentional abstraction of process logic into a separate service layer needs to be established alongside the definition of other service layers to ensure that subsequent modeling and design processes properly carry out the allocation of this logic. As with the inventory structure patterns from Chapter 5, this pattern is realized via analysis and design processes, such as those explained in Chapter 3.

Note also that while it is common to associate a task service with a single business process, this limitation is not required. As with any service, a task service can be comprised of multiple capabilities, each of which represents a separate process or task. The only rule is that these processes be related to a common overarching functional context.

Often the desire to limit a task service to a single process is related to scalability and autonomy concerns. Given that a task service will generally contain a great deal of composition logic, it is usually beneficial to limit its functional scope to a single composition so that it does not impose performance burden upon one composition when being invoked to compose another.

These types of runtime performance issues is one of the reasons this pattern is combined with Process Centralization (193) as part of the application of Orchestration (701).

Impacts

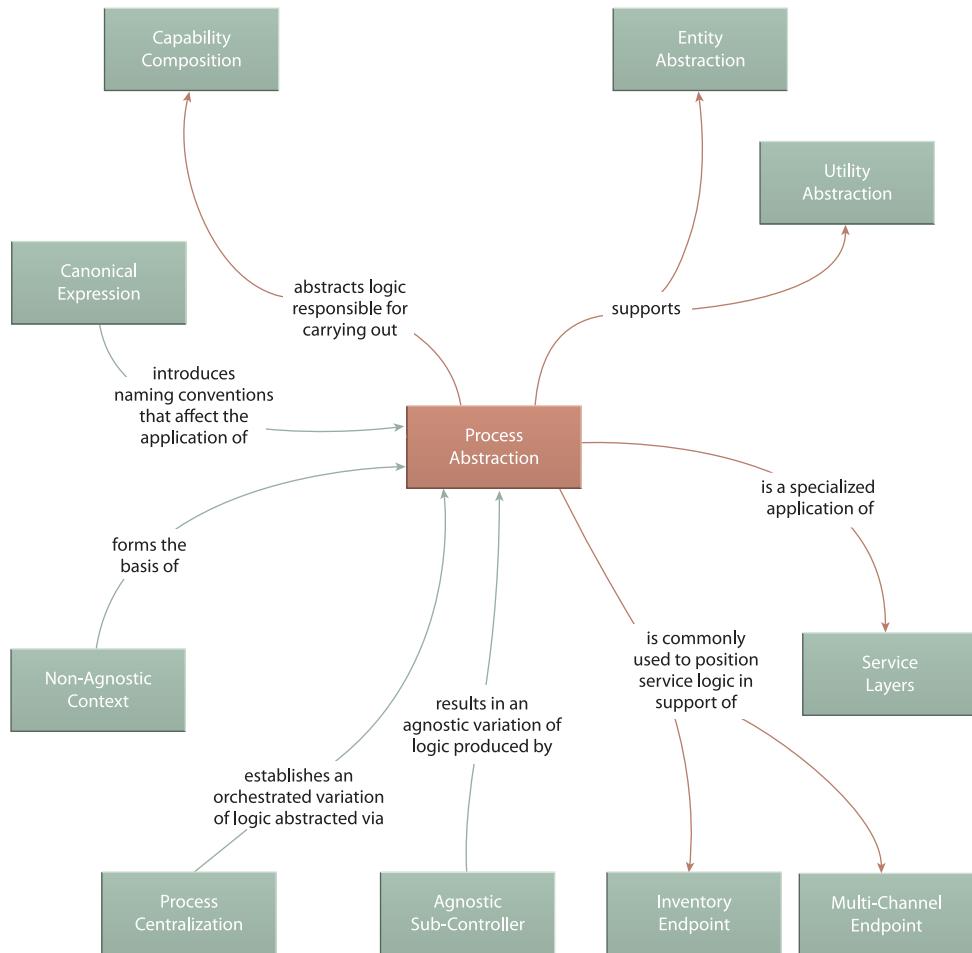
The deliberate separation of business process logic into dedicated services generally positions task services as parent controllers of service compositions. Because essential agnostic logic will have been abstracted into other services, task services will almost always depend on multiple agnostic services to carry out their business process logic. An organization needs to be prepared to implement and support service compositions in order for this pattern to be effectively applied.

Furthermore, this pattern places logic into services that could otherwise be located into other types of service consumer programs. This in itself introduces additional design and development effort.

Relationships

Because Process Abstraction provides a service classification dedicated to encapsulating non-agnostic logic, its application filters out single-purpose logic in support of defining agnostic services, as per Entity Abstraction (175) and Utility Abstraction (168).

The key foundation of this pattern is Non-Agnostic Context (319), which establishes the intentionally single-purpose scope that results in the creation of task services.

**Figure 7.13**

Process Abstraction is vital to establishing a parent business task service layer wherein single purpose logic can be placed so that agnostic services can be comprised of pure, multi-purpose logic.

Note also that Process Abstraction is a core part of Orchestration (701). The concept of abstracting parent process logic into a logical layer forms the basis for modern orchestration platforms that are commonly based on Web service composition technologies, such as WS-BPEL. In fact, when part of this compound pattern, the application of Process Abstraction results in a variation of the task service model called the *orchestrated task service*.

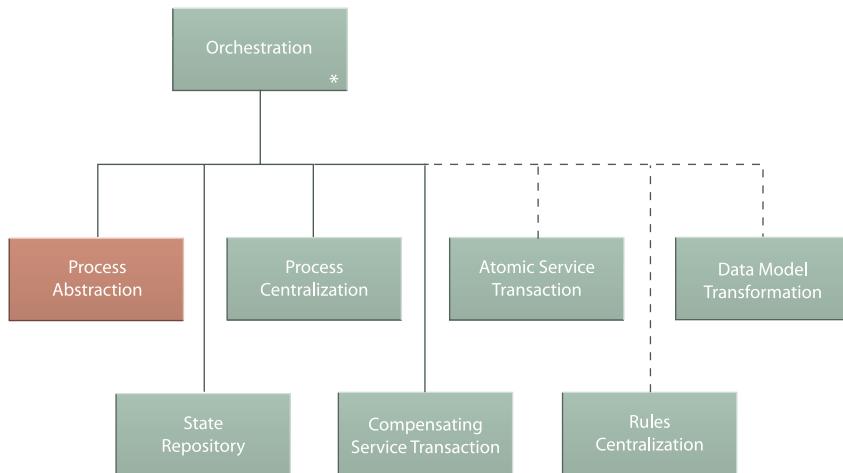


Figure 7.14

Process Abstraction is one of four core patterns that comprise Orchestration (701).

CASE STUDY EXAMPLE

There are literally hundreds of services planned as part of the FRC's long-term effort to build an enterprise service inventory. Phase one of the roll-out schedule is comprised of the delivery of a core set of services that represent fundamental functions within the organization. The goal is to establish services as an accepted functional medium within the respective divisions before moving toward large-scale, cross-divisional process automation.

Following are some of the results of a large-scale service inventory analysis effort:

- *Policies* are official documents based on a series of *regulations* governed by the FRC's Policy Management division.
- The Field Support division sends out FRC *officers* to inspect various forestry *organization sites* and issue *evaluations*.
- Based on these evaluations and other factors, the Assessment and Appeals division issues *assessment reports*.
- Negative assessments can require that organizations pay *fines*. However, these assessments can also be *appealed*.

- Various types of intra- and inter-divisional *reports* are anticipated, many of which will require runtime *conversion* of disparate data models and formats.

As with the Cutit and Alleywood examples, most of the service candidates are again based on the entity service model and the Reports and Divisional Conversion service candidates at the bottom of Figure 7.15 represent the utility service layer.



Figure 7.15

Three groups of services, each core to the business operation of an FRC division.

To understand why Adjust Policy Appeal is classified as a task service, we need to go beyond the artifact relationships to learn more about the analysis work carried out by FRC architects and analysts.

Adjust Policy Appeal is a specific business process that was decomposed and studied during the service-oriented analysis process. It essentially represents a business task whereby statistics from successful and failed appeals are collected and assessed to indicate possible changes made to various metrics (fees, date ranges, etc.) used in FRC policies. This business process requires the involvement of various previously defined services, including Policy, Appealed Assessments, and Reports.

It therefore contains logic comprised of:

- functions that can be completed by entity and utility services
- functions specific to the Adjust Policy Appeal business process, such as decision logic, composition logic, and unique calculations

Whereas the former type of functionality is carried out by agnostic entity and utility services, the latter type is specific to the Adjust Policy Appeal business process and therefore considered non-agnostic. This makes it suitable for encapsulation within a task service as part of the task service layer.

NOTE

The additionally displayed Consolidate Applications task service represents a separate business process not explained here. However, the justification for its logic being part of a task service is the same as with Adjust Policy Appeal.

This page intentionally left blank



Chapter 8

Inventory Centralization Patterns

Process Centralization

Schema Centralization

Policy Centralization

Rules Centralization

The design patterns in the preceding chapter focused on organizing a service inventory into logical domains, which means their application doesn't affect the physical location of individual services. For example, all of the services within an entity abstraction layer are not expected to reside on the same computer or server.

These next patterns, however, do address physical aspects of service inventory architecture, as follows:

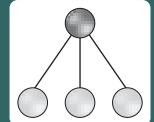
- Process Centralization (193) advocates that logic associated with different business processes should be kept in the same location.
- Schema Centralization (200) positions standardized schemas as physically independent parts of the inventory architecture so that they can be shared across services and also used independently from services.
- Policy Centralization (207) helps establish global and domain-level policies that are physically isolated and can therefore also be shared by and applied to multiple services.
- Rules Centralization (216) is focused on separating processing logic and data storage specific to the management of business rules data.

Because these are centralization patterns, each introduces some extent of inventory-wide standardization. This is an important and recurring application requirement (and impact) for all patterns in this chapter.

Furthermore, unlike the patterns in the preceding chapters which have so far been quite fundamental to inventory design, the physical inventory centralization patterns can be considered more specialized. Although they are recommended, these patterns are not absolutely required to establish a basic service inventory.

Process Centralization

How can abstracted business process logic be centrally governed?



Problem	When business process logic is distributed across independent service implementations, it can be problematic to extend and evolve.
Solution	Logic representing numerous business processes can be deployed and governed from a central location.
Application	Middleware platforms generally provide the necessary orchestration technologies to apply this pattern.
Impacts	Significant infrastructure and architectural changes are imposed when the required middleware is introduced.
Principles	Service Autonomy, Service Statelessness, Service Composability
Architecture	Inventory, Composition

Table 8.1

Profile summary for the Process Centralization pattern.

Problem

Within environments containing larger service inventories, the single-purpose requirement to concurrently support the automation of multiple business processes is common. Business process logic that spans business entities (process logic that cannot be represented by any one entity service) can be placed into individual task services (as shown in Figure 8.1).

While these services exist as peer members of a service inventory, the fact that they are independently implemented results in an enterprise's business process logic being physically distributed across multiple locations. When changes come along, the ability to efficiently extend, streamline, or even combine business process logic is inhibited because the underlying logic of each affected task service needs to be revisited, opened up, and changed, as required.

Furthermore, due to the nature of varying workflow logic, some business processes cannot be carried out in real-time. Instead, they may impose long-running service activities that can span minutes, hours, and even days. Independent task service implementations need to be equipped with state deferral extensions to facilitate these requirements. While this is

technically feasible, it can become somewhat tedious to repeat these implementation extensions across numerous individual service environments, especially when the task services are highly distributed across different physical servers (and perhaps even across different vendor runtime platforms).

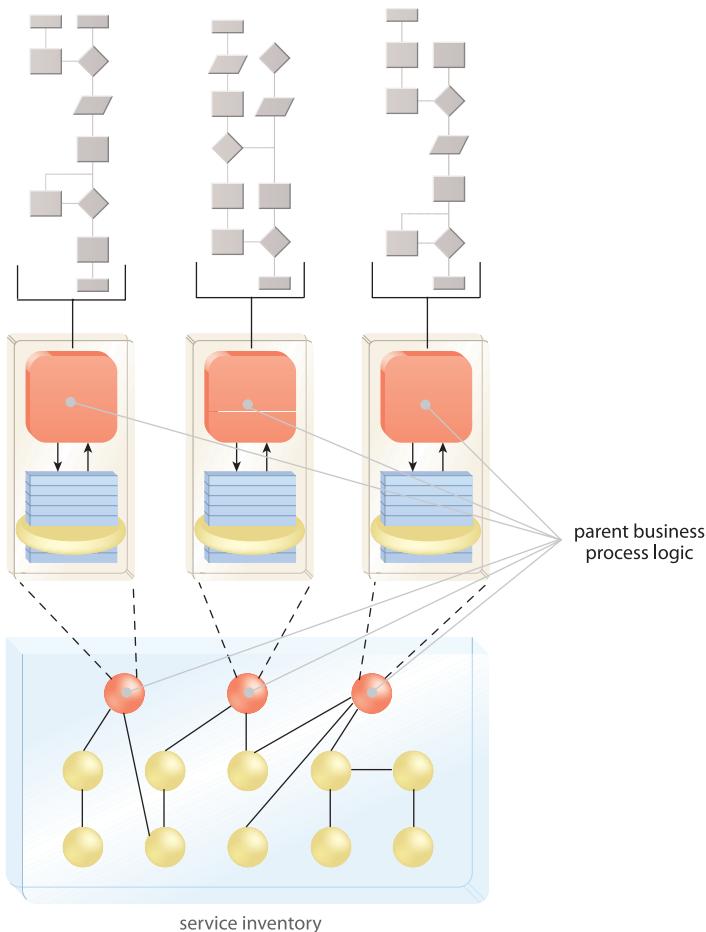
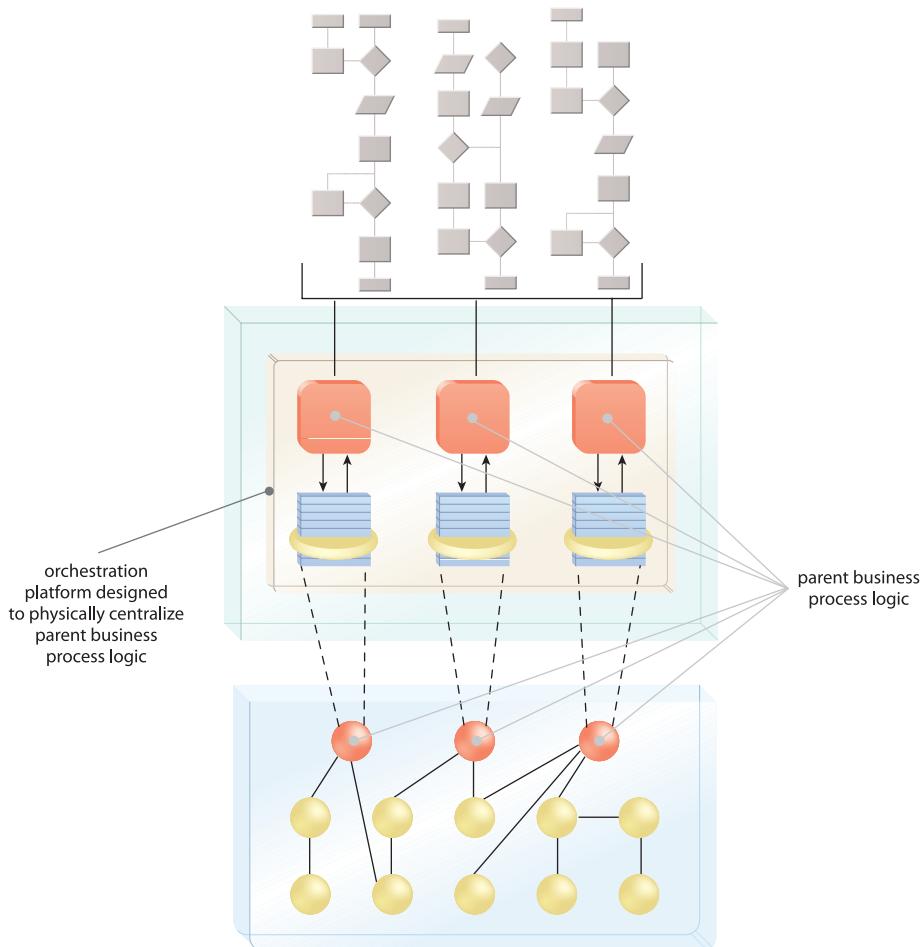


Figure 8.1

Task services are commonly implemented as individual Web services. Because each program contains embedded business process logic, it results in a physically decentralized architecture.

Solution

Parent business process logic (representing some or all of the business processes within a given domain) is centralized into one location. An orchestration platform hosts and executes this logic while allowing for its on-going, centralized maintenance (Figure 8.2).

**Figure 8.2**

Task services can continue to be implemented as separate Web services, but as part of an orchestration platform their collective business process logic is centrally located and governed (resulting in “orchestrated” task services).

Application

Modern variations of orchestration platforms that emerged during the EAI era provide a fundamental medium for centralizing process logic. When combined with support for open business process definition languages (such as WS-BPEL), these platforms become suitable for establishing a primary parent composition layer within SOA.

To realize this design pattern, a modern orchestration platform is required. Such an environment is typically comprised of the following:

- a graphical front-end tool allowing users to express and maintain business process logic
- a back-end middleware runtime environment capable of hosting orchestrated task services and the corresponding collection of business process definitions created with the front-end tool
- features that comply with industry standards related to business process logic expression and execution, such as WS-BPEL

In a nutshell, the composition logic for a specific business process is defined using the front-end tool and then encapsulated by a specific orchestrated task service. The backend platform hosts the service in the same environment as others, allowing these services to carry out their composition logic with a range of supporting features, including state management and various service agents.

NOTE

Orchestration (701) is not absolutely required to apply this pattern. Placing logic for multiple business processes into a single task service can also be considered an application of Process Centralization. However, for reasons explained in the description for Process Abstraction (182), this practice is not always recommended.

Impacts

Introducing orchestration technology into an enterprise can be expensive and disruptive. The infrastructure requirements to host and run the necessary middleware can increase the size and overall operational costs of the IT environment as a whole. It is therefore best to decide whether an orchestration layer can be established early in the technology architecture planning process.

The overall impact of this design pattern depends on the extent to which service-related middleware already exists as part of the enterprise. If no middle tier exists, its introduction will affect the surrounding infrastructure and the complexion of the overall technology architecture including existing service inventories.

Furthermore, the front- and back-end products required to support orchestration are rarely implemented in isolation. When creating an orchestration environment, the middleware platform is typically expanded to encompass a range of centralized service governance functions.

NOTE

This pattern can also be applied after a service inventory has already been established. As long as Process Abstraction (182) was used to define a layer of task services, the proper separation of agnostic and non-agnostic logic will exist to allow for the non-agnostic (process-specific) logic to be cleanly migrated to a central location.

Relationships

This pattern raises a number of architectural considerations that consequently establish relationships with a variety of patterns. Because Process Centralization represents a core part of Orchestration (701), Canonical Resources (237) comes into play, especially when more than one orchestration product is a possibility.

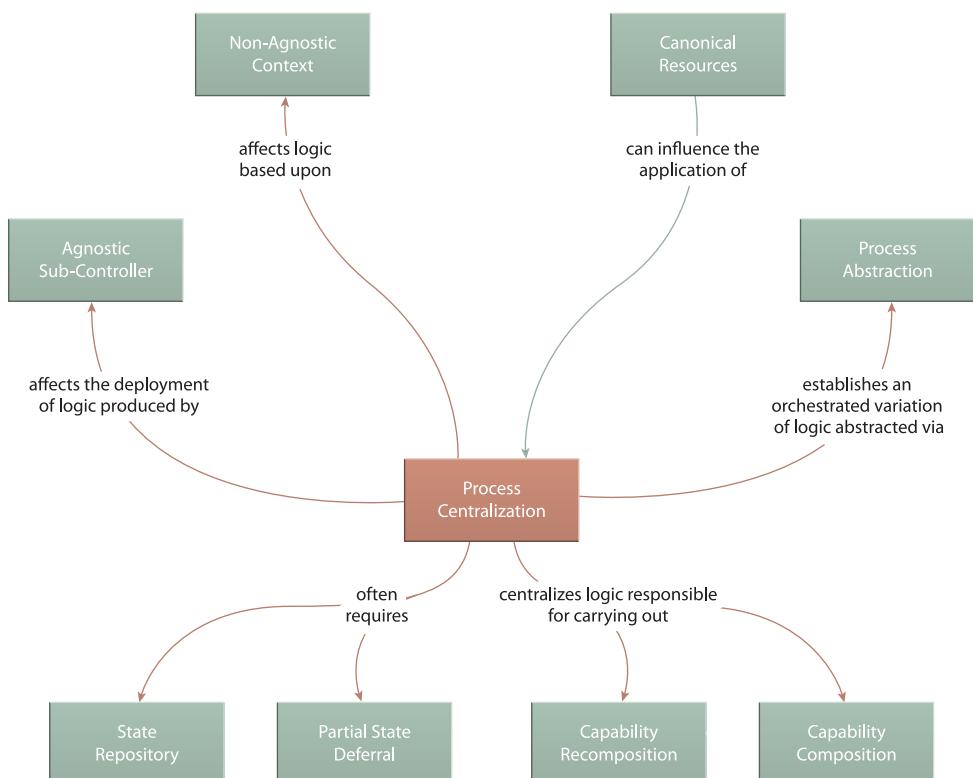


Figure 8.3

Process Centralization establishes a physical process hub within an architecture, and therefore can affect the application of several other patterns.

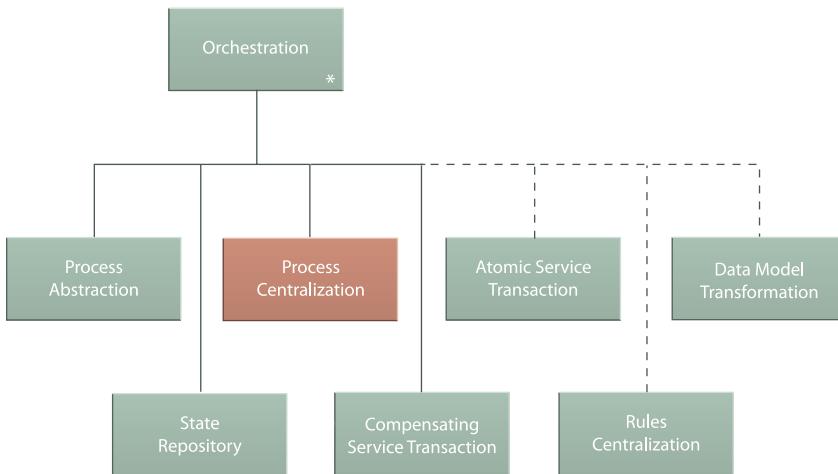


Figure 8.4

Process Centralization is most commonly associated with its role as part of the core patterns that represent Orchestration (701).

Process-centralized environments naturally require state management extensions, as per State Repository (242) and Partial State Deferral (356), due to the tendency of orchestrated task services to be more stateful and to allow for the temporary storage of state data in support of long-running process activities.

Finally, because this pattern is only focused on the physical location of process logic, it equally supports both Capability Composition (521) and Capability Recomposition (526).

CASE STUDY EXAMPLE

During a larger service modeling exercise, the McPherson Enterprise team realizes that as part the initial planned roll-out there will be the need to support six different complex service compositions within just the Tri-Fold environment alone. After completing subsequent project phases, it is estimated that the quantity of service compositions could easily triple (including compositions that will need to access services in the Alleywood inventory).

From the beginning, it was assumed that some sort of orchestration platform would be needed to establish a true enterprise middleware implementation in support of their SOA project. These latest findings appear to support this conclusion, but the team is

still uncertain as to whether to make the use of the orchestration environment mandatory for all task services (effectively turning each task service into an orchestrated task service), or whether to make this decision on an individual basis.

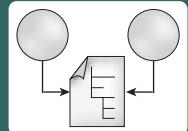
The former option would establish the equivalent of an enterprise design standard, requiring that all task logic be expressed in WS-BPEL and that it be carried out in the centralized orchestration environments.

Some team members feel that such a standard would be overkill and would not allow for the delivery of more optimized task services that would exist as standalone Web services. Others feel it could compromise attaining desired levels of Service Autonomy that have been identified for some composition requirements.

After some discussion, the decision to establish a design standard based on Process Centralization was postponed. The team agrees to give their new orchestration product a nine month period to prove that it is suitable for the range of processing requirements their compositions will need to fulfill.

Schema Centralization

How can service contracts be designed to avoid redundant data representation?



Problem	Different service contracts often need to express capabilities that process similar business documents or data sets, resulting in redundant schema content that is difficult to govern.
Solution	Select schemas that exist as physically separate parts of the service contract are shared across multiple contracts.
Application	Up-front analysis effort is required to establish a schema layer independent of and in support of the service layer.
Impacts	Governance of shared schemas becomes increasingly important as multiple services can form dependencies on the same schema definitions.
Principles	Standardized Service Contract, Service Loose Coupling
Architecture	Inventory, Service

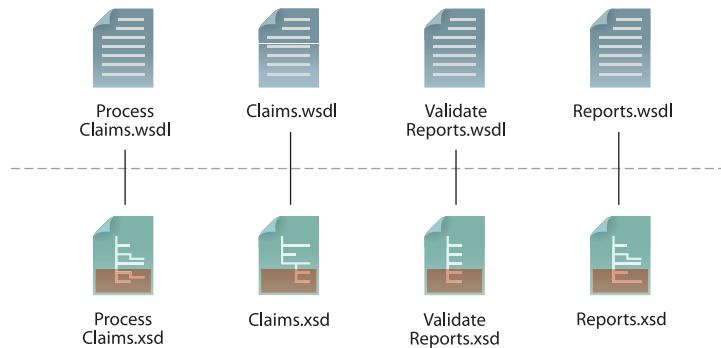
Table 8.2

Profile summary for the Schema Centralization pattern.

Problem

When building services for larger enterprise environments, the context established by each service boundary will usually not be exclusive to one body of data. For example a Claims service will represent a collection of claims-related functions and will therefore be primarily responsible for processing claims data. However, even though it will be positioned as a primary endpoint for that body of functionality, it will likely not be the only service to work with claims data.

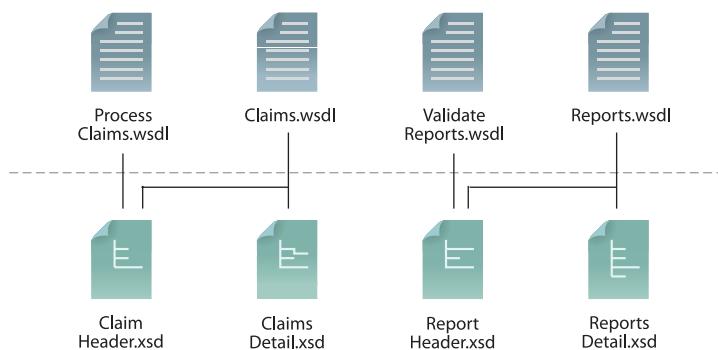
As a result, the need for duplicate schema data models emerges, leading to the definition of service contracts with redundant content (Figure 8.5). Even if the data models across these contracts are standardized, the redundant and decentralized implementation of contract schemas introduces constant governance challenges primarily associated with keeping schema data models in synch.

**Figure 8.5**

A set of WSDL definitions for which a corresponding set of XML schemas has been custom tailored. This has the appearance of a very clean contract architecture, but it actually can introduce significant schema content redundancy as indicated by the red-shaded areas.

Solution

Schemas can be designed and implemented independently from the service capabilities that utilize them to represent the structure and typing of message content. As a result, a schema architecture can be established and standardized somewhat separately from the parent service layer. For example, if one schema representing claims data is defined, any service with a capability that needs to process claims data would use the same schema (Figure 8.6).

**Figure 8.6**

WSDL definitions that share common XML schemas end up sharing the same message data models. Note how the reduction of redundant content also results in smaller-sized schemas.

Application

What is primarily advocated by this pattern is the creation of an independent schema (or data representation) architecture. This architecture may already exist within an organization, especially if serious XML Schema standardization efforts have already been carried out. However, if schemas need to be defined as part of the SOA initiative, then it is recommended that they be created prior to the completion of individual service contracts.

Ideally, the incorporation of the separate schema layer is taken into account subsequent to the completion of the service inventory blueprint, in preparation for the delivery of the physical service inventory.

The following sequence is suggested:

1. Complete a service inventory blueprint to establish a conceptual representation of planned services within an inventory.
2. Determine the required centralized schema definitions to represent the common business entities and information sets likely to be processed by services in this inventory.
3. Create the schema definitions by applying design standards to ensure consistency and normalization.
4. Create the WSDL definitions using the standard schemas wherever appropriate and supplementing the contract with any required service-specific schemas.

NOTE

Even though this design pattern advocates the avoidance of redundant schema content, in most environments it is common to supplement centralized schemas with service-specific schemas. It can be impractical and even impossible to centralize all schemas within an inventory.

Impacts

Because of the dependencies formed on the shared schema definitions, their initial design is crucial. After multiple service contracts form links to a schema, the evolution of the schema definition becomes a key part of the overall service inventory governance. Any change to a centralized schema can affect numerous service contracts.

For larger organizations, this level of data standardization can pose daunting challenges, many of which revolve around the maintenance of the shared schemas and the enforcement of associated design standards.

Relationships

The schema layer established by Schema Centralization can be built upon and further incorporated as part of a Contract Centralization (409) effort and, due to its emphasis on reducing data model redundancy, also carries forward the goals of Service Normalization (131) into the data tier.

An interesting relationship can exist between this pattern and Validation Abstraction (429) in that the creation of official, centralized schemas can bring with it more detail than all services may actually require. This can end up countering some of the optimization and information hiding goals of Validation Abstraction (429).

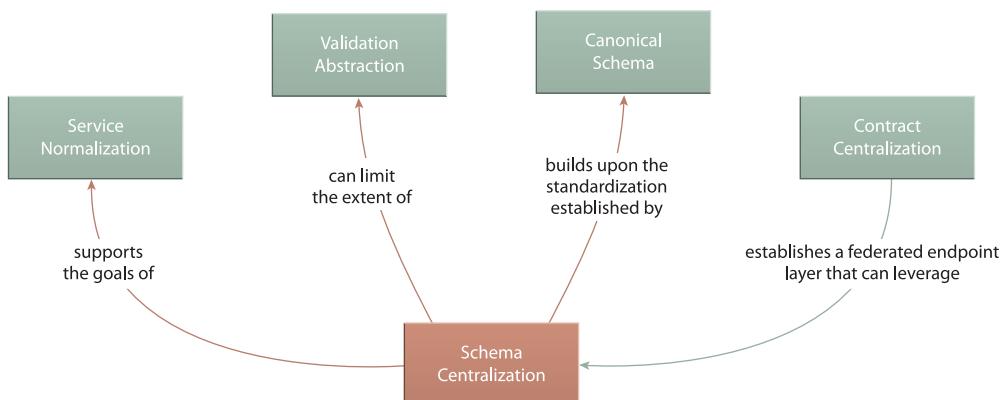


Figure 8.7

Schema Centralization has few relationships because its scope is limited to an independent, underlying data model tier.

CASE STUDY EXAMPLE

The processing logic behind the four Cutit services is reviewed by architects in an effort to identify where common data sets are used.

The following areas are found:

- The Run Chain Inventory Transfer, Inventory, and Chain services are required to process data associated with a chain manufacturing record.
- The Run Chain Inventory Transfer, Inventory, and Order services are required to process data associated with a chain inventory record.
- The Run Chain Inventory Transfer and Order services are required to process order records.

As a result, the project team employs data analysts to create standardized data models (as XML schemas) for chain manufacturing, chain inventory, and order record documents, as shown in Figure 8.8.

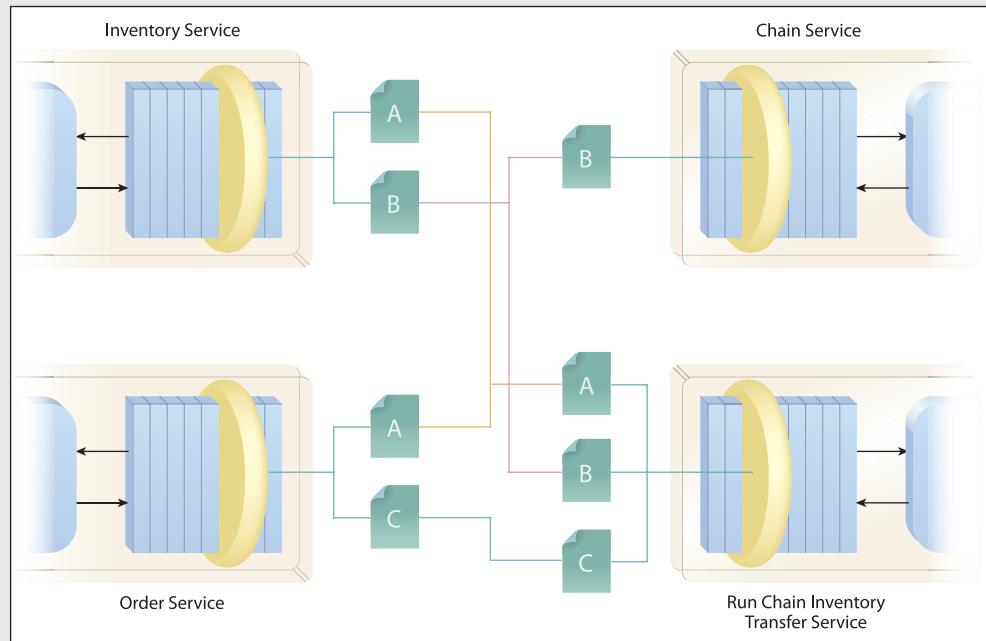


Figure 8.8

The XML schemas (A, B, C) defined for the individual Cutit Web services are synchronized wherever common data sets are identified. (A represents the inventory record, B represents the chain manufacturing record, and C represents the order record.)

The Cutit team now decides to streamline the overall data representation architecture by avoiding redundant schema content. The result is a service contract structure similar to what is displayed in Figure 8.8, only central physical schemas are shared across services (Figure 8.9).

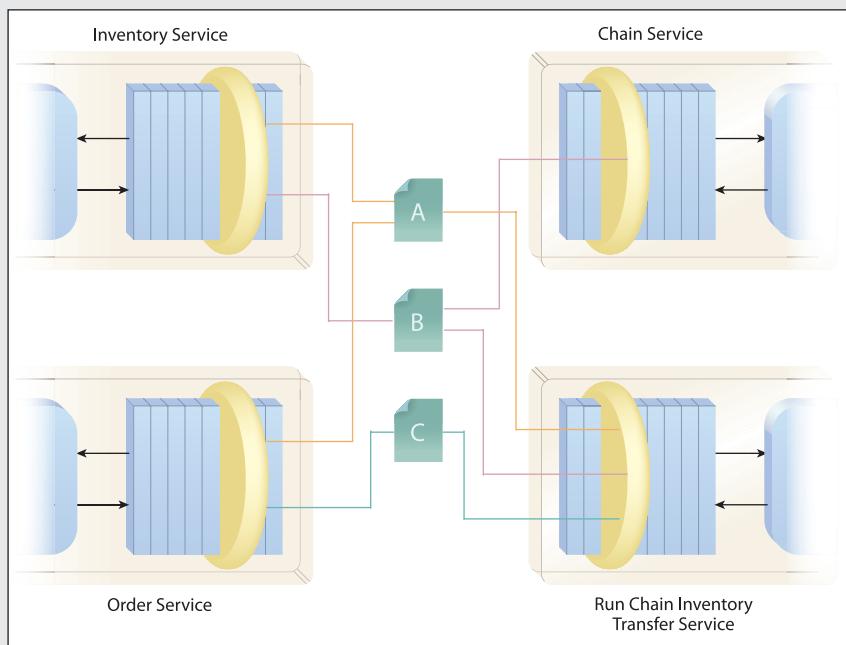


Figure 8.9

Schemas A, B, and C are centrally positioned and linked to via the service contracts that are required to process the corresponding data sets. (Additional service-specific schemas still exist but are not shown in this figure.)

Assuming Schema A in the preceding diagram is `inventory.xsd`, and Schema C is `order.xsd`, the WSDL definition for the Order service might begin as follows:

```
<definitions targetNamespace=
  "http://cutitsaws.com/contract/order"
  xmlns:tns="http://cutitsaws.com/contract/order"
  ...>
<types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:import namespace=
      "http://cutitsaws.com/schema/inventory"
      schemaLocation=
      "http://cutitsaws.com/schema/inventory.xsd"/>
    <xsd:import namespace=
      "http://cutitsaws.com/schema/order"
      schemaLocation=
      "http://cutitsaws.com/schema/order.xsd"/>
  </xsd:schema>
```

```
...  
</types>  
...  
</definitions>
```

Example 8.1

The Order service contract imports the centralized inventory and order schemas, which are also shared by other service contracts.

NOTE

The application of this pattern is further explored in Chapter 6 of *SOA Principles of Service Design* as part of the case study example and in Chapter 14 of *Web Service Contract Design and Versioning for SOA*.

Note also that the *Web Service Contract Design and Versioning for SOA* book establishes a convention with regards to the spelling of the term “XML Schema.” The word “schema” is capitalized when referring to the XML Schema language or specification and it is lower case when discussing schema documents in general.

For example, the following statement makes reference to the XML Schema language:

“One feature provided by XML Schema is the ability to...”

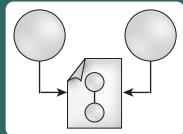
And this sentence explains the use of XML schema documents:

“When defining an XML schema it is important to...”

This spelling convention is also used in this book.

Policy Centralization

How can policies be normalized and consistently enforced across multiple services?



Problem	Policies that apply to multiple services can introduce redundancy and inconsistency within service logic and contracts.
Solution	Global or domain-specific policies can be isolated and applied to multiple services.
Application	Up-front analysis effort specific to defining and establishing reusable policies is recommended, and an appropriate policy enforcement framework is required.
Impacts	Policy frameworks can introduce performance overhead and may impose dependencies on proprietary technologies. There is also the risk of conflict between centralized and service-specific policies.
Principles	Standardized Service Contracts, Service Loose Coupling, Service Abstraction
Architecture	Inventory, Service

Table 8.3

Profile summary for the Policy Centralization pattern.

Problems

Services may be required to process a variety of individual policies (also called *policy expressions*). Areas commonly addressed by policies include security and transaction requirements, as well as a variety of quality-of-service (QoS) properties.

Regulatory policies may affect a range of services, whereas other policies may be service-specific. A service built as a Web service can establish policy requirements as part of its contract via the use of WS-Policy expressions, or it may apply policies within its underlying service logic.

When common policies are repeated across multiple service contracts, they introduce redundancy into the service inventory (Figure 8.10). This leads to bloated policy content and increases the governance burden required to ensure that common policies are kept in sync over time.

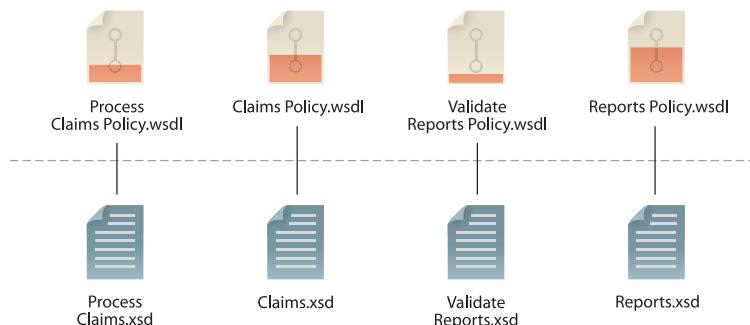


Figure 8.10

Each of the four displayed WSDL documents is extended with individual WS-Policy definitions. The red shading indicates the extent of redundancy across policies.

Solution

Policies that apply to multiple services can be abstracted into separate policy definition documents or service agents that are part of an inventory-wide policy enforcement framework. Abstracted policies can be positioned to apply to multiple services, thereby reducing redundancy and providing centralized policy governance (Figure 8.11).

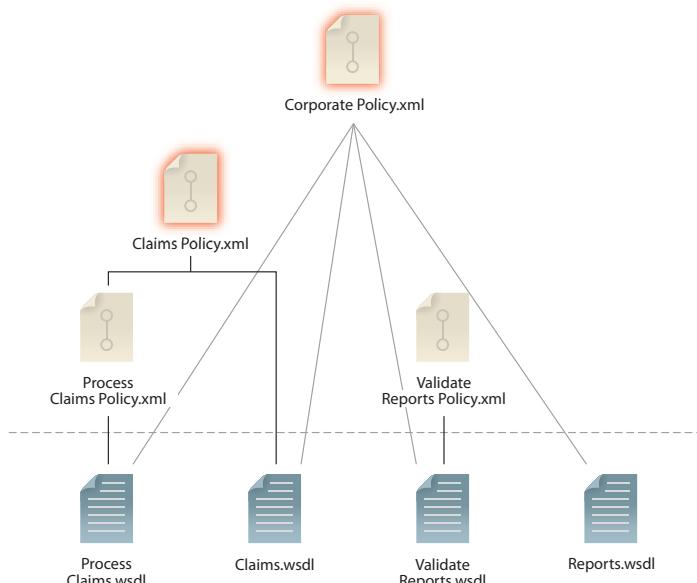


Figure 8.11

A global policy definition (Corporate Policy.xml) is established and applies to all WSDL documents, and a further domain policy (Claims Policy.xml) is created and linked to two WSDL definitions. This new policy structure eliminates redundancy of policy content and ensures consistent policy enforcement.

Application

A policy framework needs to be added to the inventory architecture so that policies can be separately defined and associated with services and then validated, enforced, and even audited at runtime.

The WS-Policy framework includes a separate WS-PolicyAttachments specification that explains binding mechanisms for policies. Policy definitions can be embedded within or linked to WSDL documents. To apply this pattern one or more policies typically need to be grouped together into a policy definition that is made available so that Web service contracts to which the policies apply can add the appropriate references.

Middle-tier platforms (such as those provided by ESB products) can provide policy features supported by runtime agents that carry out policy compliance checking. In these environments, global and domain-level policies can also be established via service agents that act as *policy enforcement points* (Figure 8.12).

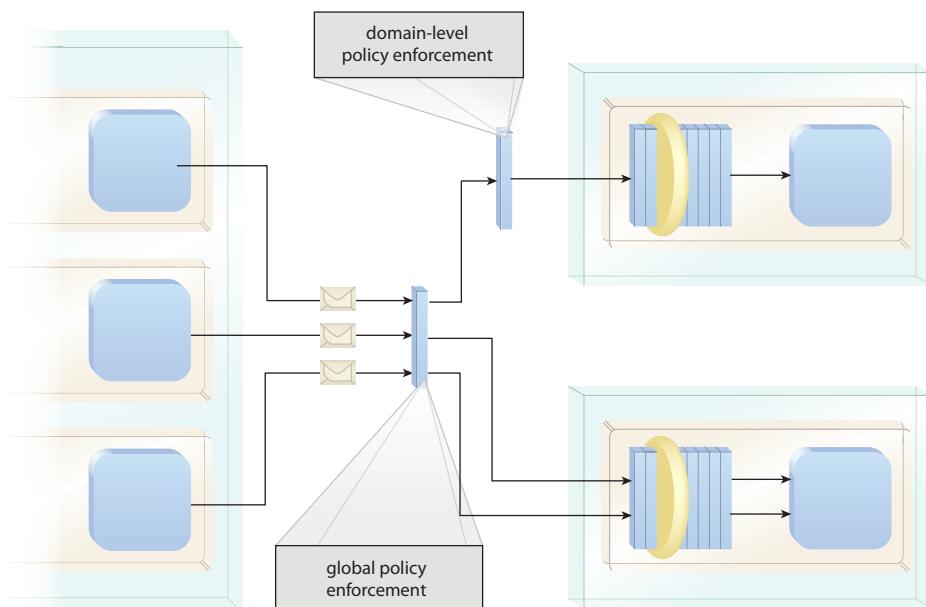


Figure 8.12

Service agents, as part of an inventory-wide policy framework, intercept incoming messages to check for policy compliance. One agent (bottom) enforces a global policy that applies to all services, while the second agent (top) enforces a domain-level policy after global policy compliance was confirmed.

NOTE

Policy requirements are often collected early on during the service delivery lifecycle. For example, service modeling processes allow analysts to determine potential policies while the service is still in the conceptual stage. Policy logic is typically documented within the service profile where policies can be defined on a service or capability level.

Impacts

Policy definition is an additional step in the service delivery lifecycle that needs to be taken into consideration early on. Part of the analysis involved with defining policies is determining what types of policies should *not* be made part of the technical service contract.

Some policies may be subject to unexpected change and therefore more likely to demand new contract versions. Other policies may be more suited for a service-level agreement (SLA) that exists as a document used by humans (usually the owners of potential consumer programs). Therefore, the Service Abstraction principle is a key factor in ensuring that the constraint granularity of contract capabilities remains reasonable.

Once global or domain-level policies are established, they need to be maintained with a great amount of care. One change or addition to a shared policy will affect all services that rely upon it. This, in turn, affects all service consumers that have formed dependencies on the corresponding service contracts. A governance structure is therefore required, comprised of assigned roles (such as policy custodians) and processes that ensure that common policies are properly evolved.

Furthermore, increased up-front analysis is required prior to the delivery of WS-Policy definitions so that policies are designed with the right balance of constraints and flexibility to accommodate the range of contracts that may be required to use them. A common problem when working with centralized policies is that conflicts can arise between policies at different levels. For example, a new global policy may inadvertently contradict a service-level policy for a particular service. Formal analysis and governance processes can help avoid these situations.

Additionally, the service agents and proxies that establish the policy enforcement points within the inventory architecture can add performance overhead and independent failure modes, which the surrounding infrastructure needs to be able to accommodate. Each centralized policy effectively adds a layer of runtime processing and service dependency.

Finally, when implementing a policy framework based on the use of service agents (as part of an ESB product, for example), it is relatively common for the WS-Policy standard to not be fully supported. Instead, the framework may require that policies be defined via front-end tools that output a proprietary policy format. Once deployed, this can lead to undesirable vendor lock-in scenarios that counter the objectives of the vendor-neutral architecture characteristic (explained in Chapter 4). The use of proprietary policy formats further can prevent inter-organization data exchange unless both organizations happen to be using the same products.

Relationships

Because Policy Centralization essentially establishes an independent policy layer that extends service contracts, it directly relates to and benefits from Contract Centralization (409).

This pattern continues the concepts established by Service Normalization (131) in that it avoids redundancy across policies via centralization, and because Policy Centralization can affect the content of a service contract, there is a further relationship with Validation Abstraction (429).

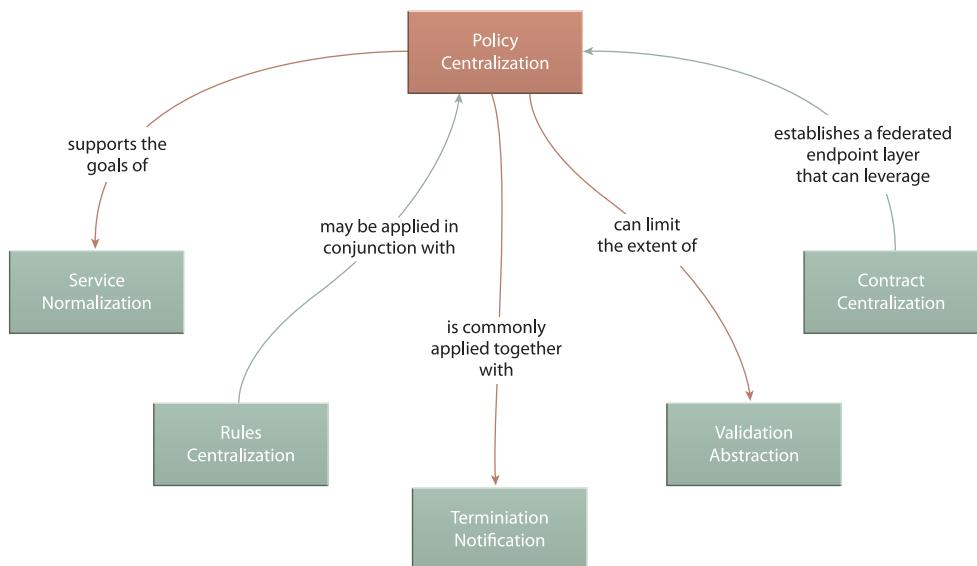


Figure 8.13

Policy Centralization positions policies within a service inventory architecture and therefore affects other patterns that either relate to the service contract layer or to the nature of policy logic.

NOTE

The nature of policy logic can vary, but the fact that policies are often based on security regulations can also tie the application of this pattern to several of the security patterns provided in Chapters 13 and 20.

ESB products have been credited with popularizing policy enforcement and the concept of centralized policies, which is why this pattern is one of the common extensions to Enterprise Service Bus (704).

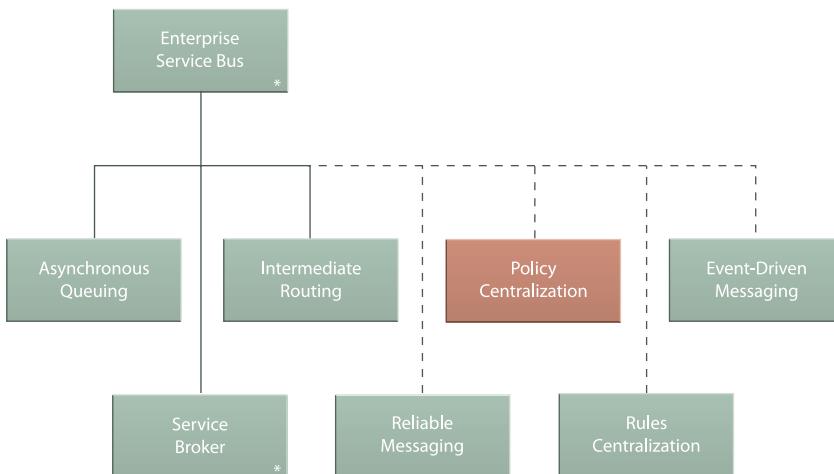


Figure 8.14

When supported, the addition of Policy Centralization brings an important layer of quality assurance to the base messaging, routing, and brokerage patterns that comprise Enterprise Service Bus (704).

NOTE

Another policy-related pattern that was developed for this book but not included in this edition is Canonical Policy Vocabulary. This pattern establishes standardized policy vocabularies required when customizing policies for use within a service inventory. You can learn more about this pattern at SOAPatterns.org.

CASE STUDY EXAMPLE

When finalizing the service contracts described in the Schema Centralization (200) case study example, Cutit architects incorporate a policy that requires that all messages transmitted to any Web service comply to the SOAP 1.2 standard.

The initial approach was to add this policy to each individual Web service contract, as shown in Figure 8.15.

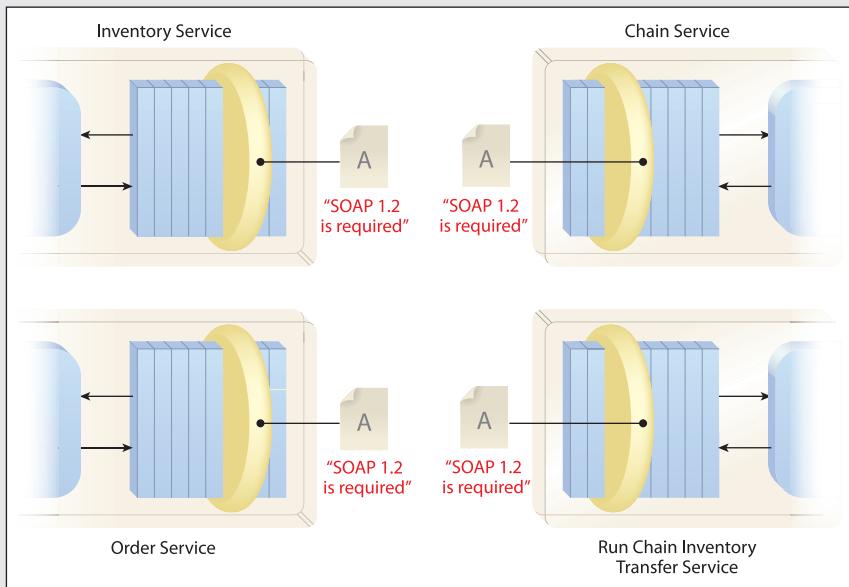


Figure 8.15

The same policy (A) is added redundantly across all Web service contracts.

After the architecture design specification was reviewed, concerns were raised about the redundancy introduced by adding identical policies across multiple contracts. Should the policies ever need to be augmented or removed, it would require a significant governance effort, especially if this approach was taken with all of the services in the Cutit inventory.

Subsequent to some research, a Cutit architect discovers that the middleware product they were considering would allow them to leverage the ability to centralize a policy so that it could be shared across multiple Web service contracts. A prototype is assembled with the architecture illustrated in Figure 8.16, demonstrating a single policy being dynamically applied to multiple Web service contracts.

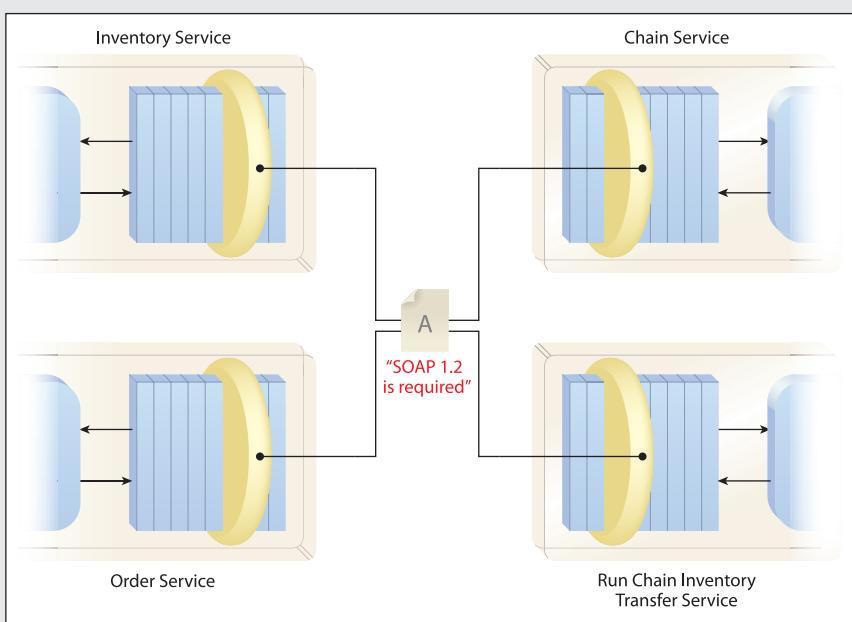


Figure 8.16

A single global policy (A) is established, thereby replacing the redundant policy definitions entirely.

Assuming Policy A in the preceding diagram resides in `globalPolicies.xml`, the `portType` element in the WSDL definition for the Order service might contain a `wsp:PolicyURIs` attribute, as follows:

```
<definitions targetNamespace=
  "http://cutitsaws.com/contract/order"
  xmlns:tns="http://cutitsaws.com/contract/order"
  ...>
  ...
  <portType name="ptOrder"
    wsp:PolicyURIs="pol:globalPolicies.xml">
    <operation name="SubmitOrder">
      <input message="tns:msgSubmitOrderRequest" />
      <output message="tns:msgSubmitOrderResponse" />
    </operation>
    ...
  </portType>
  ...
</definitions>
```

Example 8.2

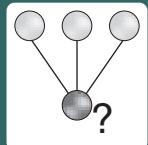
The Order service contract with an external reference to a global policy definition that is shared by other services.

NOTE

Besides the use of the `wsp:PolicyURIs` attribute, there are several other ways to externally reference and attach policies to WSDL definitions, as explored in Chapter 16 of *Web Service Contract Design and Versioning for SOA*.

Rules Centralization

How can business rules be abstracted and centrally governed?



Problem	The same business rules may apply across different business services, leading to redundancy and governance challenges.
Solution	The storage and management of business rules are positioned within a dedicated architectural extension from where they can be centrally accessed and maintained.
Application	The use of a business rules management system or engine is employed and accessed via system agents or a dedicated service.
Impacts	Services are subjected to increased performance overhead, risk, and architectural dependency.
Principles	Service Reusability
Architecture	Inventory

Table 8.4

Profile summary for the Rules Centralization pattern.

Problem

The workflow logic within any given business process is driven by and structured around rules specific to how the logic must be carried out, as per the policies, regulations, and preferences of the organization. Individual business service capabilities frequently must carry out their encapsulated logic in accordance with these rules.

It is not uncommon for the same rule to be applied to different scenarios involving different business entities. This results in a need to incorporate one rule within multiple bodies of service logic. As an organization changes over time, so do certain business rules. This can lead to modifications within individual entity business services as well as business process logic encapsulated by task services or otherwise (including the occasional utility service). Having to revisit multiple services each time a business rule changes can be counter-productive.

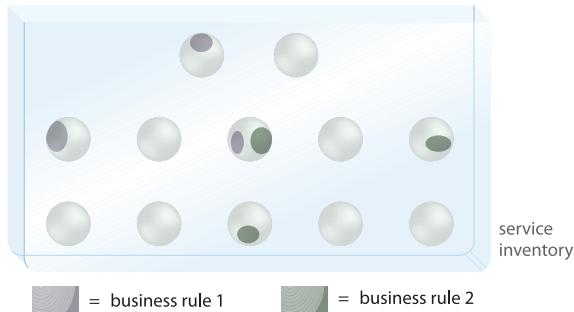


Figure 8.17

Just two business rules can find their way into several different business services and, in this case, even a utility service. A global change to either rule will therefore impact multiple services.

Solution

Business rules can be physically abstracted into a dedicated part of the architecture under the management of specialized rules engines and platforms. This centralizes access to business rule logic and avoids redundancy. It further centralizes the governance of business rules so that they can be modified and evolved from a single location.

Application

Different business rules management systems exist, each introducing a relatively proprietary runtime and administration platform. A central service can be established to provide an official access point for the creation, modification, retrieval, and application of business rules.

Modern runtime platforms also offer native rules repositories and processing logic that is made accessible via a set of system service agents and APIs. This allows any service to interface with business rules-related logic without having to compose a separate service.

NOTE

Centralized rule services are most often classified as members of the utility service layer because they provide generic processing functionality that leverages technology resources and because their functional context is not derived from any organization-specific business models. Even though rule data is business-centric, to the rules service it is just data that it is required to manage and dispense.

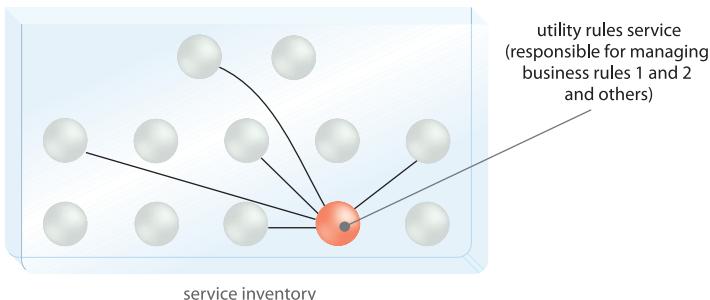


Figure 8.18

All business rules are encapsulated by a single rules service accessed at runtime by other services that need to retrieve or apply business rule logic. (Service agents are also commonly used to provide native access to abstracted rules, as explained shortly.)

Impacts

Because this pattern is applied across an entire service inventory, it can impact an architecture in several ways:

- While it achieves the centralization of business rules data within an inventory, Rules Centralization also ends up *decentralizing* business logic associated with business services. For example, business rules related to the processing of invoices would normally be encapsulated by an Invoice entity service. However, this pattern would move those business rules into a separate location.
- The performance requirements of affected services are increased due to the need for business rules to be retrieved or applied at runtime. Caching mechanisms can alleviate this impact to an extent (usually when rules are temporarily stored as state information for a particular service composition).
- If existing runtime platform features cannot be leveraged to establish centralized rules management, this pattern generally results in the introduction of a separate business rules management product. This extension can increase the size, complexity, and overall operational cost of a technology architecture and must furthermore be sufficiently reliable to consistently accommodate service usage patterns. A rules management system prone to runtime failure can paralyze an entire service inventory.

- Accessing centralized business rules via native system agents and APIs will impose tight architectural dependencies upon services. If many business services use these runtime features, the overall service inventory could become “locked in” to a particular vendor platform.
- Because the actual business rule logic is physically separated, the scope of logic encapsulated by several business service capabilities is incomplete (as per their parent contexts), and their overall autonomy is decreased.

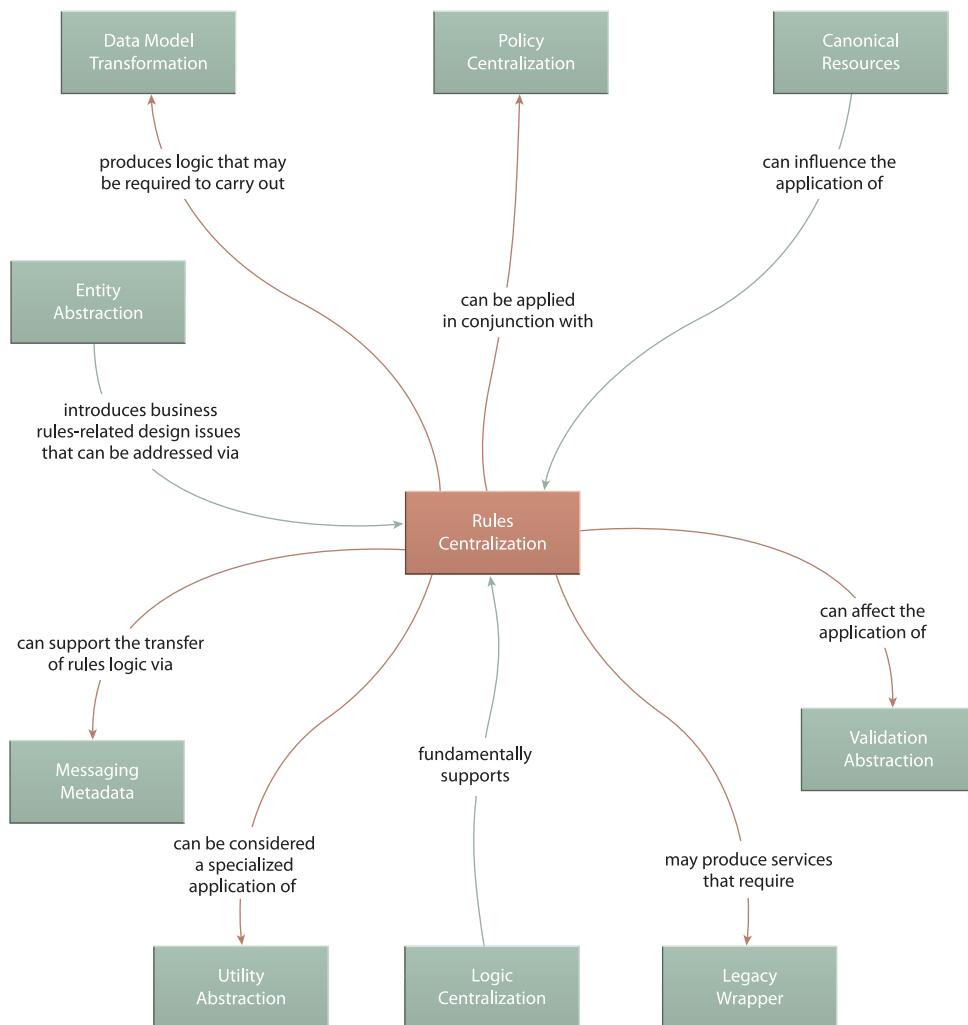
Another issue worth noting is the actual management of centralized business rules. Often a technical administrator is in charge of the rules system, but multiple business domain experts may be needed to maintain the business rules themselves. This can lead to ownership challenges in that the custodian of a business service must also be involved with the maintenance of a subset of the business rules within the central rules repository in addition to the governance of the business service itself.

Relationships

Business rules can be found just about anywhere within a typical service-oriented solution, which is why the abstraction and centralization of rules data can affect the content of a service contract, as per Validation Abstraction (429).

Because this pattern may result in the creation of specialized rules utility services, it is naturally related to Agnostic Context (312) and Utility Abstraction (168), as well as Cross-Domain Utility Layer (267). As a reusable utility service, a rules service may need to encapsulate proprietary rules engines or products, which can lead to the need for Legacy Wrapper (441) and which also ties into the regulatory influence of Canonical Resources (237). Finally, policies will often need to incorporate or introduce rules, which is why this pattern may be applied together with Policy Centralization (207).

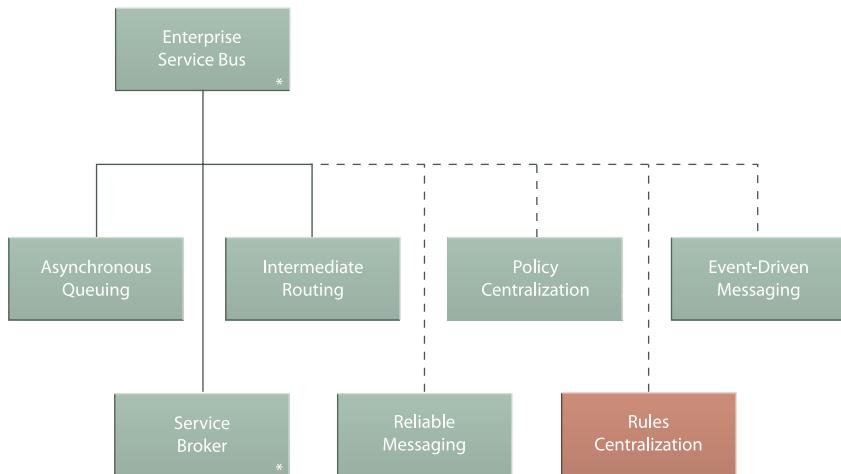
Centralized business rules are commonly leveraged to increase the sophistication with which ESB products carry out messaging, routing, and brokerage-related functions. In ESB environments, the variation of this pattern resulting in native agents and APIs is more common than the creation of dedicated rules services. Similarly, this pattern can be leveraged by Orchestration (701) so that business rule logic can be incorporated into workflow and composition logic.

**Figure 8.19**

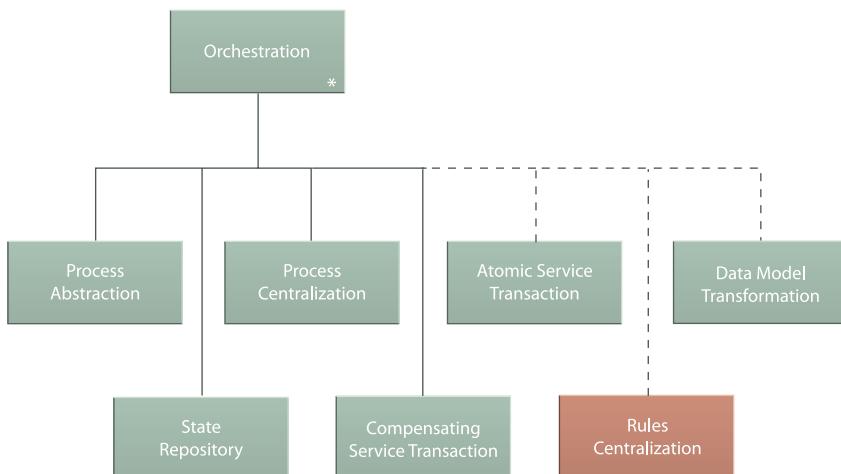
The Rules Centralization pattern establishes utility logic that can affect the application of a variety of other patterns.

NOTE

In the upcoming case study example the repeated references to policies relate to regulatory policies, as opposed to technical policies that were the focal point of the preceding case study example for Policy Centralization (207).

**Figure 8.20**

One of the optional parts of Enterprise Service Bus (704) is that of native Rules Centralization, allowing much of the core ESB functions to be driven by business rule logic.

**Figure 8.21**

Business rules processing can also be part of an orchestration environment, which is why this pattern is considered an optional extension of Orchestration (701).

CASE STUDY EXAMPLE

The FRC manages a large amount of policies that regulate the commercial forestry industry. Different policies apply to different types of forestry companies, but ultimately, many of the policy rules and requirements are inter-related. If one policy needs to be changed, then that change can affect a series of other policies that are in some way connected or dependent.

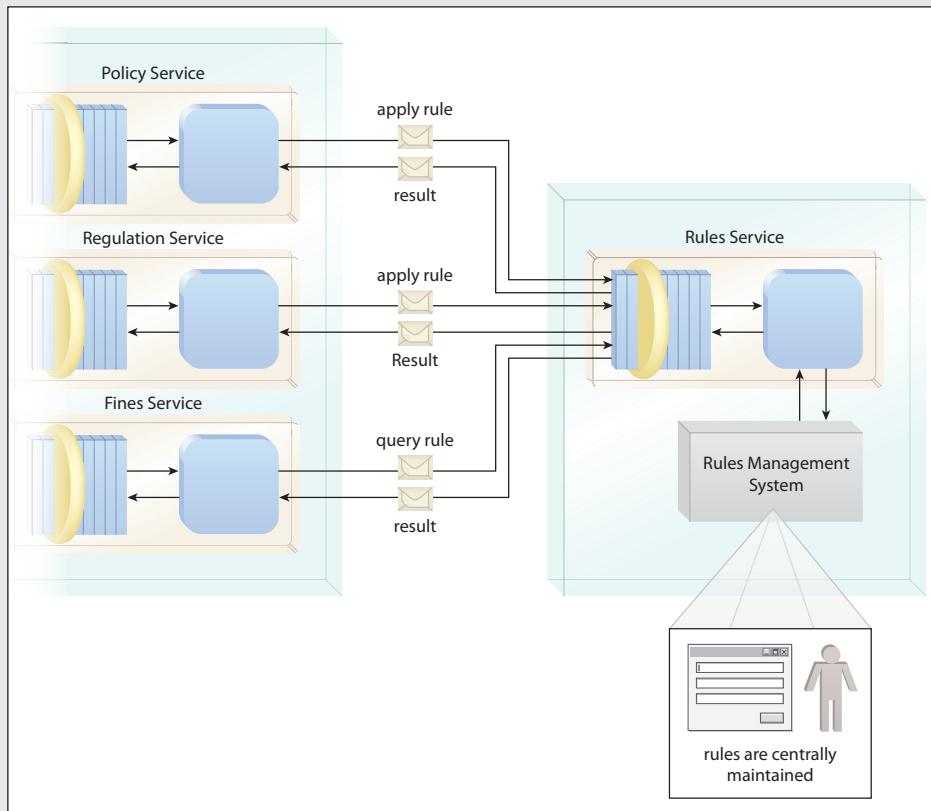


Figure 8.22

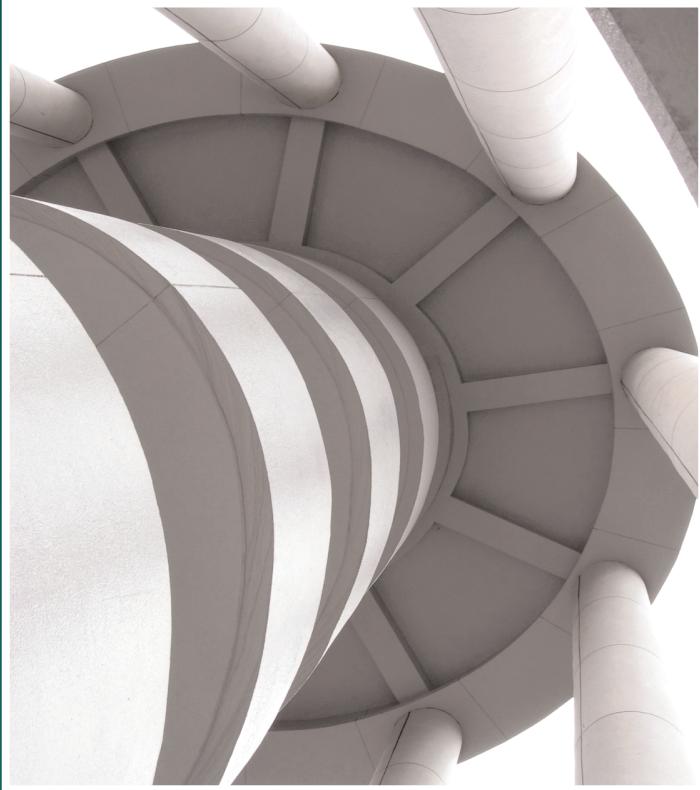
The Rules service encapsulating a proprietary rules management system provides central access to rules-related processing logic for other services. The rules management product also centralizes all business rules data for maintenance by a dedicated administrator.

Within the FRC service inventory there are many services (mostly entity-centric) that require access to policy business rules. During the early service modeling stages, it was determined that these rules should not be managed by these services individually. Doing so would result in an unacceptable amount of logic redundancy.

Because so many of the policies tied back into a core set of business rules, it was deemed necessary to establish a service responsible for the management, issuance, and application of policy-related business rules. This Rules service was classified as a utility service and became a central part of their inventory.

This page intentionally left blank

Chapter 9



Inventory Implementation Patterns

Dual Protocols

Canonical Resources

State Repository

Stateful Services

Service Grid

Inventory Endpoint

Cross-Domain Utility Layer

To address further common design issues relating to service inventory architecture, this chapter provides a set of specialized patterns that help solve implementation-level problems.

Each of these five patterns targets a specific area of inventory architecture:

- Dual Protocols (227) provides a flexible solution that addresses the challenges of establishing a canonical communications protocol.
- Standardization of underlying technologies is advocated by Canonical Resources (237).
- State Repository (242) and Stateful Services (248) provide alternative solutions for runtime state data deferral.
- Service Grid (254) proposes a sophisticated solution for state deferral and fault tolerance.

The following additional two patterns are focused on solving extra-inventory architectural concerns for environments in which multiple domain inventories exist or for when communication external to the inventory boundary needs to be accommodated:

- Inventory Endpoint (260) establishes somewhat of a specialized proxy service that interacts with external consumers on behalf of services within the inventory boundary.
- Cross-Domain Utility Layer (267) proposes a design solution that changes the face of domain inventories by stretching a common layer of utility services across inventory boundaries.

Whereas the objective of Inventory Endpoint (260) is to preserve the integrity of services within a boundary at the cost of increasing logic redundancy, the goals behind Cross-Domain Utility Layer (267) are to open up portions of these boundaries for the purpose of reducing enterprise-wide redundancy and increasing reuse.

NOTE

Some of the patterns in this chapter reference the term “service activity.” Be sure to revisit the definition in Chapter 3 if the term is not familiar to you.

Dual Protocols



How can a service inventory overcome the limitations of its canonical protocol while still remaining standardized?

Problem	Canonical Protocol (150) requires that all services conform to the use of the same communications technology; however, a single protocol may not be able to accommodate all service requirements, thereby introducing limitations.
Solution	The service inventory architecture is designed to support services based on primary and secondary protocols.
Application	Primary and secondary service levels are created and collectively represent the service endpoint layer. All services are subject to standard service-orientation design considerations and specific guidelines are followed to minimize the impact of not following Canonical Protocol (150).
Impacts	This pattern can lead to a convoluted inventory architecture, increased governance effort and expense, and (when poorly applied) an unhealthy dependence on Protocol Bridging (687). Because the endpoint layer is semi-federated, the quantity of potential consumers and reuse opportunities is decreased.
Principles	Standardized Service Contract, Service Loose Coupling, Service Abstraction, Service Autonomy, Service Composability
Architecture	Inventory, Service

Table 9.1

Profile summary for the Dual Protocols pattern.

NOTE

For a definition of what the term “protocol” refers to in this pattern, see the *What Do We Mean by “Protocol?”* section in the pattern description for Canonical Protocol (150).

Problem

As advocated by Canonical Protocol (150), it is preferred for all services within an inventory to interact using the same communications technology. However, when inventory-wide protocol standardization is not possible or when the chosen communications technology is inadequate for certain types of data exchanges, it can compromise service interoperability, thereby undermining the overall goals of Canonical Protocol (150).

Solution

Two levels of services are delivered within the same inventory:

- a primary level based on the preferred protocol
- a secondary level based on an alternative protocol

This allows the secondary protocol to be used whenever the primary protocol is deemed deficient or inappropriate. This solution furthermore allows services based on the secondary protocol to be promoted to the primary protocol when appropriate.

Application

A popular example of a transport plus messaging protocol combination that is chosen for standardization but that is part of a technology platform that may not be suitable for all types of services is SOAP over HTTP. Even though services built as Web services can establish a standardized communications framework based on these technologies, this choice can raise some issues.

For example:

- SOAP introduces message-processing overhead that may be unreasonable for service capabilities that need to exchange granular amounts of data or that need to be invoked multiple times by the same consumer during the same business process.
- The additional messaging-related processing may be considered inappropriate for services that physically co-exist on the same server and do not require remote communication.
- The service may require a special feature that cannot be accommodated by the Web services technology platform due to an absence of vendor support or a gap or deficiency in a supported Web service standard.

As stated earlier, issues such as these can make it difficult to justify Canonical Protocol (150) on an inventory-wide basis.

Dual Protocols therefore provides a compromise that is essentially based on the standardization of two canonical protocols. For example, when applying this pattern to a Web services-based service inventory, services built as Web services are typically classified as the primary service level because the use of Web services supports several other design benefits and patterns that leverage its industry standards.

However, for circumstances where Web services do not represent a suitable implementation option for services, a secondary protocol is chosen (Figure 9.1). Most commonly, this alternative protocol is based on a particular component platform (such as Java or .NET). In this case, components are designed as self-contained services subject to the full set of service-orientation design principles (including the standardization of the component interface via the Standardized Service Contract principle).

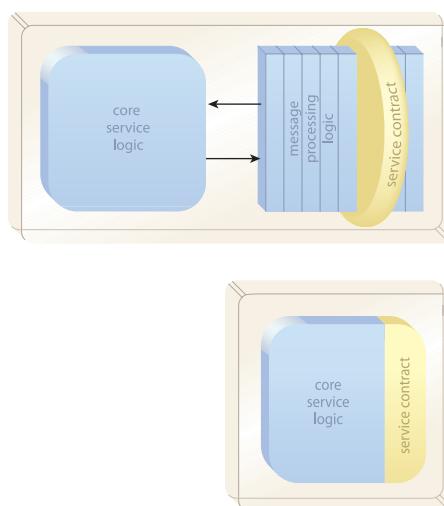


Figure 9.1

A service implemented as a Web service (top) and a service implemented as a component (bottom). Both have standardized service contracts and are subject to all service-orientation design principles.

Figure 9.2 illustrates how primary services existing as Web services can co-exist with secondary services existing as components. Both primary and secondary service levels represent official endpoints as part of a semi-federated service endpoint layer.

There are significant risks when applying this pattern as explained in the upcoming *Impacts* section. To minimize this impact, the following guidelines are recommended:

- Contract Centralization (409) must always be respected, which means that services based on the primary protocol must be accessed via the primary protocol when invoked by secondary services. In the case of Web services, this requires that component-based services not directly access the underlying components or resources of Web services-based services.
- Consider some or all services in the secondary level as transition candidates. If this pattern was chosen due to a lack of maturity in the primary protocol, then secondary services can be earmarked for an upgrade to the primary level once the technology has sufficiently evolved.
- During a transitional period, use Concurrent Contracts (421) to enable a service to be accessible via either protocol. This way, it can begin to interoperate using the primary protocol while continuing to support consumers that rely upon the secondary protocol.
- Apply Redundant Implementation (345) wherever feasible in support of secondary services. This is especially relevant when component-based secondary services are primarily composed by the core service logic of Web services-based services to avoid remote communication. Redundant Implementation (345) will support the autonomy of both primary and secondary service levels.

Note that some secondary services may never transition and therefore always remain based on the secondary protocol. This may be due to the nature of their functionality or the convenience of keeping them for intra-service composition purposes only.

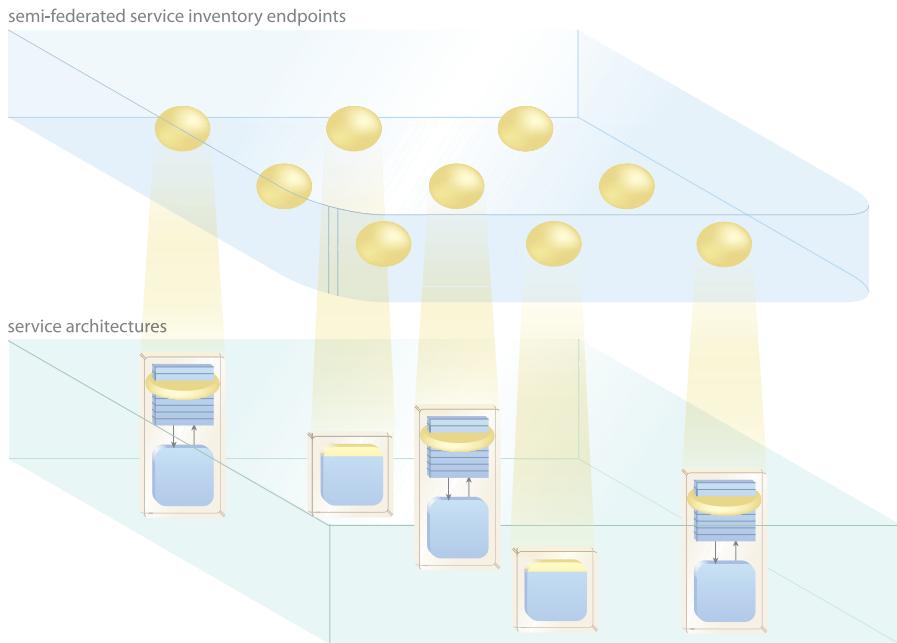
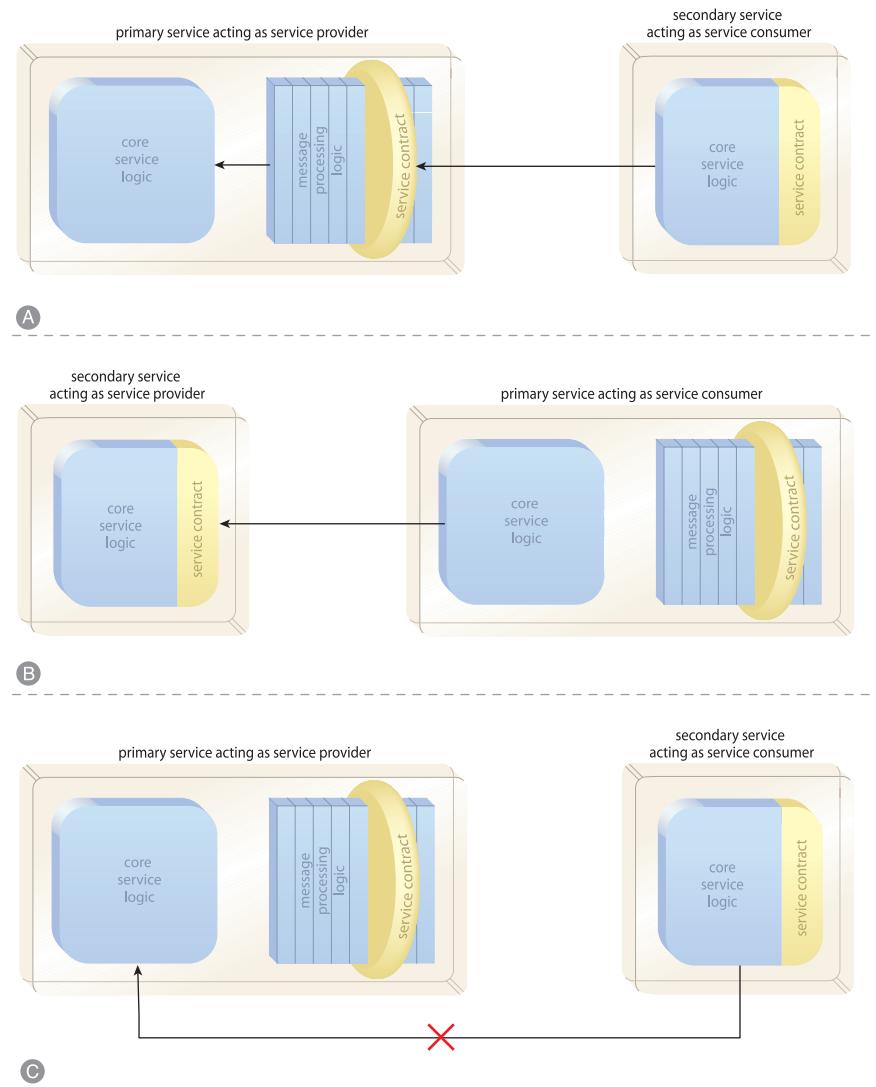


Figure 9.2

From an endpoint perspective the service contracts are all standardized, but their implementations are based on different communication protocols.

As shown in Figure 9.3, the first guideline in the previous list establishes some ground rules as to how primary and secondary services should and should not interact.

The key requirement to successfully applying this pattern is for services to continue adhering to Standardized Service Contract, Service Loose Coupling, and Service Abstraction design principles to avoid many of the negative, indirect coupling types that can lead to governance problems.

**Figure 9.3**

Regardless of protocol, all services must invoke each other via their official service contracts (A, B). Bypassing the contract may seem convenient when the underlying service logic of the primary service supports the same protocol as the secondary service (C), but it is an anti-pattern that will eventually inhibit the application of this pattern and further weaken the overall service inventory foundation.

NOTE

An alternative approach to applying this pattern is to limit the secondary protocol to utility services only. When working with Web services as the primary protocol and a native component technology as the secondary protocol, this approach can reduce the size of Web service compositions by limiting them to business services. These business Web services can then compose component-based utility services, as required.

Although this pattern description is focused on components and Web services as implementation mediums, REST services and the use of HTTP as an application protocol provide another viable option. To learn more, visit SOAPatterns.org and read up on the REST-inspired patterns currently in development.

Impacts

This design pattern must be used in moderation. It imposes some significant architectural constraints and sacrifices that need to be carefully assessed before committing to an architecture based on primary and secondary protocols.

For example:

- The use of Concurrent Contracts (421) to provide secondary services with two interfaces while they are being transitioned from secondary to primary status can lead to overly complex governance requirements. If this pattern is applied to a large service inventory with a large percentage of secondary services, the transition effort may be unwieldy.
- The repeated application of Redundant Implementation (345) in support of secondary services can rapidly increase infrastructure budgets and the overall configuration management effort required to keep all deployments of a given service in synch.
- Depending on which technologies are chosen for primary and secondary protocol levels, this pattern may limit the application of other key design patterns, such as Canonical Schema (158) and Schema Centralization (200).
- The examples in this chapter were focused on Web services comprised of components that shared the same protocol technology as the component-based services. If this pattern is applied to primary and secondary service levels that are based on disparate protocols, it will introduce the need for the constant application of Protocol Bridging (687).

- This pattern introduces the on-going risk of imposing too much technology coupling upon consumers, thereby making plans to migrate to a fully federated service inventory difficult to fully attain.

There are concrete benefits to carrying out this design pattern in the right way, but it introduces a whole new dimension to a service-oriented architecture adoption, and the associated risks need to be planned for in advance.

Relationships

The extra requirements that come with applying Dual Protocols often need to be addressed with the application of additional supporting patterns, such as Redundant Implementation (345), Concurrent Contracts (421), and Protocol Bridging (687).

Although this pattern fundamentally preserves the goals of Logic Centralization (136) and Contract Centralization (409), it ends up augmenting the default approach of carrying out Canonical Protocol (150) by essentially allowing two canonical protocols.

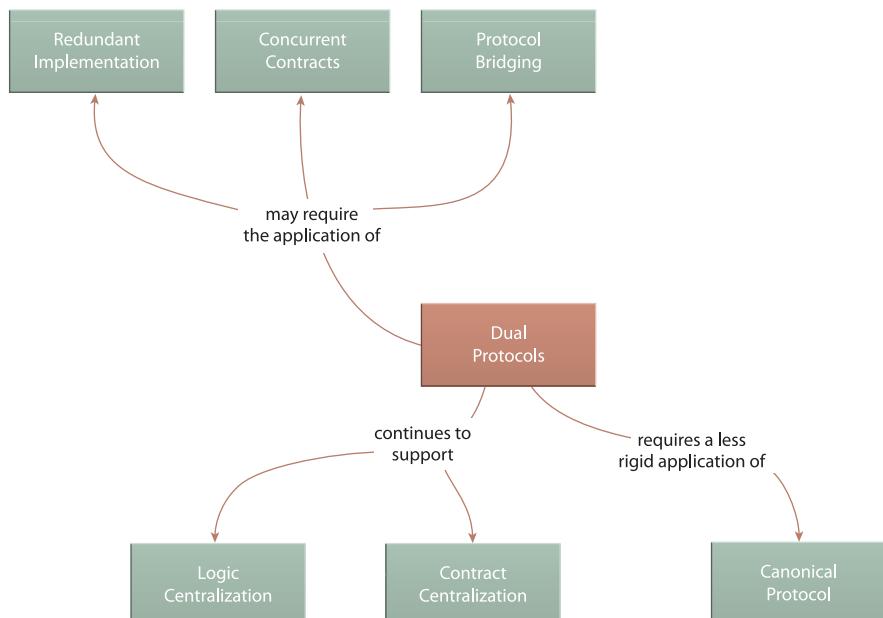


Figure 9.4

Dual Protocols can increase the flexibility and complexity of an inventory architecture and therefore requires the support of other key patterns.

CASE STUDY EXAMPLE

The Field Support office at the FRC has relied on a custom-built, distributed Web application for over five years. This system is relatively out of date by today's standards and is primarily comprised of a series of COM components and Web service scripts deployed across the traditional three physical tiers: Web server, application server, and a dedicated database server. The DCOM protocol that was popular during the 1990s is still used to communicate between the Web and application servers.

Although the IT group that has been maintaining this system is being mandated to support a broad adoption of Web services, they will not receive funding for another two years to complete this transition. Yet, in the meantime, they are still required to make parts of their system (including database access) available via SOAP messaging from other FRC applications.

To accomplish this, they first consider simply deploying a set of Web services that act as endpoints into their environment. This would satisfy immediate requirements without too much up-front effort or investment.

However, upon further discussion with architects from FRC's EA department, they begin to realize that the Web services they would be delivering would not conform to standardized schemas and would therefore not properly represent the business services within the Field Support division. They would essentially just be integration endpoints.

In the long-term, many consumer programs could form dependencies on these services, thereby entrenching their contracts. When this department is ready to move over to a full-scale services architecture, the "real" services that would then be modeled would be incompatible with these endpoints.

As a result, they would either have to disrupt the existing connections by replacing the original Web services with properly modeled ones, or they would need to classify those Web services as a legacy part of their environment that would then need to be further wrapped within newly standardized services, as per Legacy Wrapper (441). Neither option is desirable.

To avoid this situation, they decide to proceed with a preliminary service inventory architecture that supports two standard communication protocols: DCOM and SOAP. A service inventory blueprint is created for their environment, and a specific subset of the modeled service candidates is chosen for initial delivery. Some, providing new internally needed functionality, will still be delivered as COM components, while others

(especially those with which external applications need to communicate) will be built as Web services.

A strategic plan is put in place, allowing services comprised of COM components to gradually transition to Web services via component technology upgrades and the incorporation of Web service contracts.

Canonical Resources

How can unnecessary infrastructure resource disparity be avoided?



Problem	Service implementations can unnecessarily introduce disparate infrastructure resources, thereby bloating the enterprise and resulting in increased governance burden.
Solution	The supporting infrastructure and architecture can be equipped with common resources and extensions that can be repeatedly utilized by different services.
Application	Enterprise design standards are defined to formalize the required use of standardized architectural resources.
Impacts	If this pattern leads to too much dependency on shared infrastructure resources, it can decrease the autonomy and mobility of services.
Principles	Service Autonomy
Architecture	Enterprise, Inventory

Table 9.2

Profile summary for the Canonical Resources pattern.

What Do We Mean by “Resource”?

Within the context of this pattern, a resource refers to an extension of the infrastructure that provides general processing functions.

Examples include:

- databases, directories, and data warehouse products
- state deferral mechanisms (such as a standard state database, standard tables within a database used for temporary storage, or grid technology)
- security processing extensions (such as a central directory or a standardized set of security technologies and/or processing agents)
- activity management extensions (such as context and transaction management frameworks)
- reliability extensions (such as a sequence-based messaging framework)

Note that a resource may or may not be shared. Note also that this pattern does not advocate sharing resources.

Problem

Services delivered without architectural design standards or developed outside of an organization (as part of an outsourced project, for example) run the risk of introducing disparate yet still redundant infrastructure resources. This can bloat an inventory architecture and unnecessarily introduce complexity, leading to increased administration and operational costs and other governance burdens associated with maintaining a bloated enterprise environment (Figure 9.5).

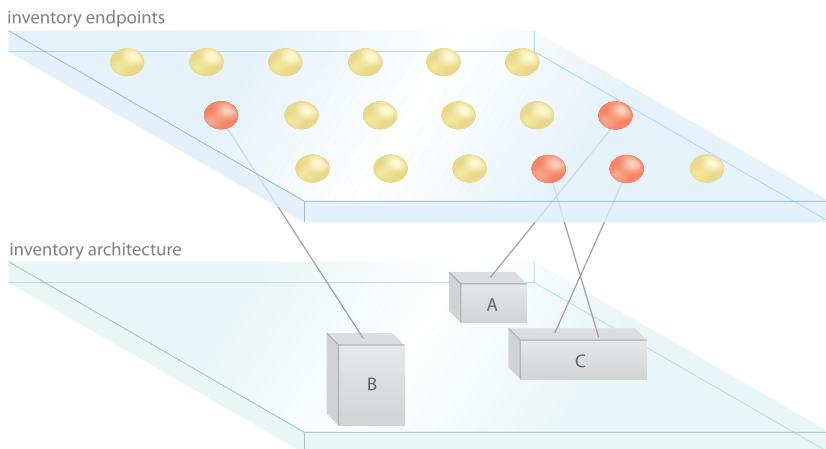
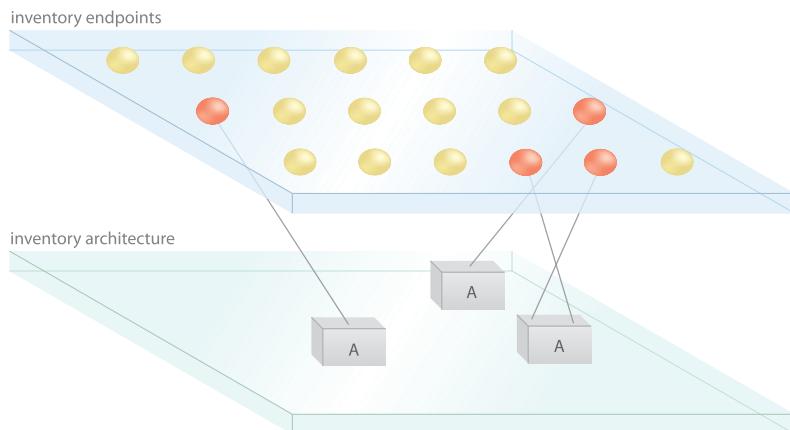


Figure 9.5

Services use different infrastructure resources for the same purpose, resulting in inconsistent architectural dependencies. In this diagram, A, B, and C could represent disparate databases or other out-of-the-box products from different vendors.

Solution

Utility Abstraction (168) is often used to wrap common infrastructure resources and then make them available via a standardized contract to the rest of the service inventory. When this is not possible, common resources are identified and standardized in order to maintain consistency across service designs and throughout the inventory in general (Figure 9.6).

**Figure 9.6**

Services use the same standardized infrastructure resource for the same purpose. Note, however, that they do not share the same implementation of the resource.

Application

This pattern is specifically focused on infrastructure products, platforms, and extensions (collectively referred to as “resources”) that provide common features useful to multiple services. These infrastructure-centric resources are essentially identified and standardized.

It is important to not allow the application of this pattern to inhibit the Vendor-Neutral design characteristic (introduced in Chapter 4) of a service inventory architecture. Therefore, the nature of the design standards that result from this pattern is preferably such that the chosen resource becomes the default option for a given requirement or purpose. This leaves the flexibility for alternatives to be considered if requirements exist that cannot be adequately fulfilled by the standardized resource.

Impacts

The repeated application of this pattern can lead to a natural tendency to want to share and reuse standardized products for cost or development efficiency purposes. This may often be warranted, but it can also inadvertently reduce the autonomy of services beyond what it should be.

Relationships

This pattern relates to others primarily as a regulatory influence. Design patterns that implement new architectural resources or extensions are encouraged to avoid introducing disparate infrastructure-related products and technologies that fulfill the same overall purpose. This affects all of the patterns listed at the top of Figure 9.7.

The end result of applying Canonical Resources is similar to enforcing an enterprise design standard, which is why Canonical Protocol (150) can be viewed as a variation of this pattern focused only on communication technologies.

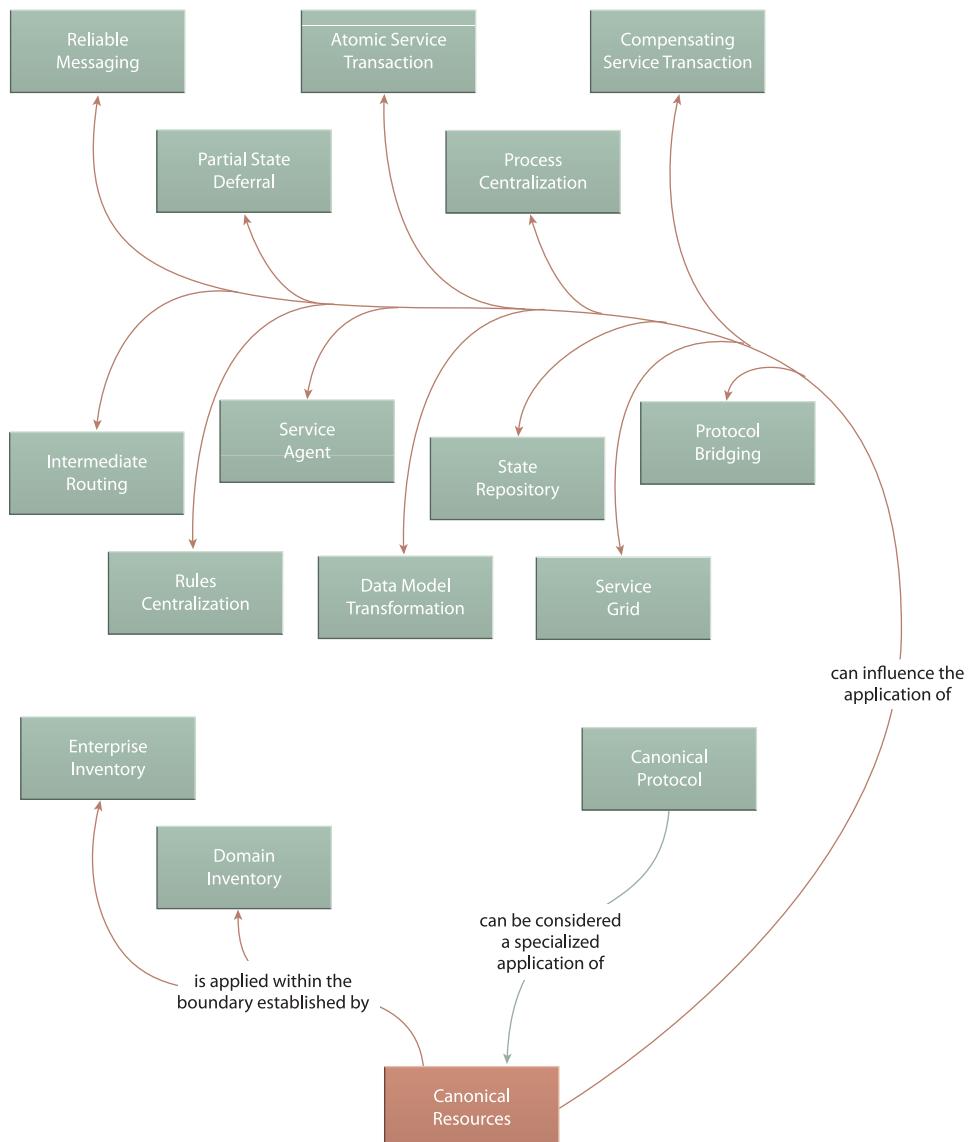


Figure 9.7

Canonical Resources helps standardize the underlying inventory architecture and therefore influences the application of many other architectural patterns.

CASE STUDY EXAMPLE

As the FRC service inventory continues to grow, more sophisticated service compositions can be assembled to automate larger, more complex business processes. Some of these tasks require an increased level of integrity to ensure that if any one composition member fails, all of the activity carried out to that point can be reversed.

This need for cross-service transactions originally inspired a solution comprised of custom SOAP headers combined with a proprietary third-party product that introduced a series of service agents to process the headers and manage the overall transaction.

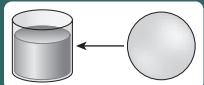
During a subsequent runtime platform upgrade, support for the WS-Coordination and WS-AtomicTransaction standards was provided, enabling ACID-style transactions to span multiple Web services, as per Atomic Service Transaction (623). New compositions leveraged this to establish an industry-standard transaction management system whereby participating services could issue standardized messages in order to vote on the outcome of the overall transaction.

However, when services using the proprietary transaction management product had to be combined with recently delivered services into new compositions, there was an evident incompatibility that required significant reworking to overcome. Essentially, the old services had to be upgraded in order to support both styles of transaction management.

It became clear that transaction management, as an architectural extension, had to become standardized across the inventory. The WS-Coordination and WS-Atomic-Transaction standards were chosen for this purpose.

State Repository

How can service state data be persisted for extended periods without consuming service runtime resources?



Problem	Large amounts of state data cached to support the activity within a running service composition can consume too much memory, especially for long-running activities, thereby decreasing scalability.
Solution	State data can be temporarily written to and then later retrieved from a dedicated state repository.
Application	A shared or dedicated repository is made available as part of the inventory or service architecture.
Impacts	The addition of required write and read functionality increases the service design complexity and can negatively affect performance.
Principles	Service Statelessness
Architecture	Inventory, Service

Table 9.3

Profile summary for the State Repository pattern.

Problem

It is often necessary to retrieve and cache bodies of data to which service capabilities require repeated access during the course of a service activity. However, some complex compositions introduce extended periods of processing during which this data is not required. While idle, this cached data continues to be stored in memory and consumes runtime resources (Figure 9.8).

This excess consumption can severely compound during periods of high concurrent usage, depleting the overall available runtime service. As this occurs repeatedly with different services throughout an inventory, overall scalability thresholds can decrease.

	pre-invocation	begin participation in activity	pause in activity participation	end participation in activity	post invocation
active + stateful		●	●	●	
active + stateless	●				●

Figure 9.8

During the lifespan of a service instance it may be required to remain stateful and keep state data cached in memory even as its participation in the activity is paused. (The orange color is used to represent the state data.)

Solution

A state repository is established as an architectural extension made available to any service for temporary state data deferral purposes (Figure 9.9). This alleviates services from having to unnecessarily keep state data in memory for extended periods.

	pre-invocation	begin participation in activity	pause in activity participation	end participation in activity	post invocation
active + stateful		●		●	
active + stateless	●		●		●
state data repository	●	●	●	●	●

Figure 9.9

By deferring state data to a state repository, the service is able to transition to a stateless condition during pauses in the activity, thereby temporarily freeing system resources.

NOTE

See the *Measuring Service Statelessness* section in Chapter 11 of *SOA Principles of Service Design* for a detailed description of state data and additional scenarios involving state data repositories.

Application

Typically, a dedicated database is provided for state deferral purposes. The database is located on the same physical server as the services that will be utilizing it, so as to minimize runtime performance overhead associated with the writing and retrieval of the data. Another approach is to create dedicated tables within an existing database. Though less effective, this still provides a state deferral option suitable for temporary data storage.

Alternatives to State Repository include Stateful Services (248) and State Messaging (557), which can be considered especially when the state data does not need to be persisted over long periods of time. However, it is also fairly common for State Repository to be used in conjunction with these patterns to provide more flexible (albeit more complex) state management mechanisms that may be especially suitable for providing customized state deferral options for different types of state data.

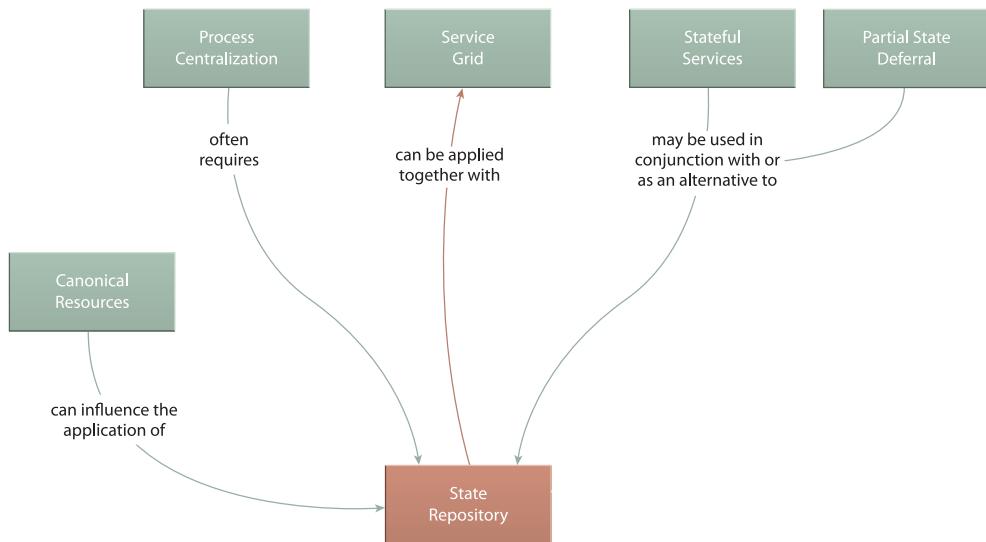
Impacts

Incorporating the state deferral logic required to carry out this pattern can increase service design complexity, leading to more development effort and expense.

Although State Repository can improve scalability, having to write data to and retrieve data from a physical hard drive generally imposes more runtime performance overhead than having to carry out the same functions against data stored in memory. For service activities with strict real-time performance requirements, this state deferral option needs to be carefully assessed.

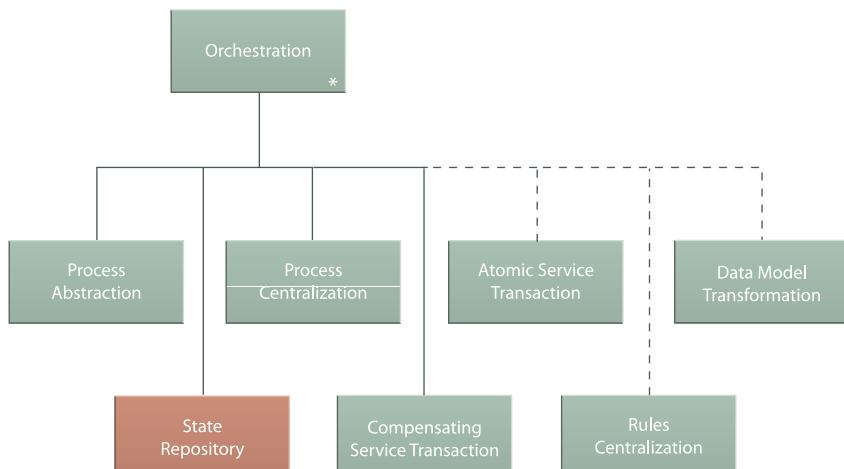
Relationships

Establishing a state management system via State Repository naturally relates to other state deferral-related patterns, such as Stateful Services (248), Partial State Deferral (356), State Messaging (557), and Service Grid (254). All of these patterns may end up using the central state database introduced by this pattern. Canonical Resources (237) can further help ensure that no one inventory will have more than one type of state management database unless absolutely required.

**Figure 9.10**

State Repository is fundamental to just about any state management design considerations and related patterns.

Process Centralization (193) will almost always require the application of this pattern to provide a means of persisting state data associated with the many business processes that orchestration environments are required to execute and manage (especially in support of long-running processes). This is why State Repository is one of the core patterns that comprise Orchestration (701).

**Figure 9.11**

State Repository is a fundamental part of the compound pattern Orchestration (701).

CASE STUDY EXAMPLE

The Alleywood Policy Check service (Figure 9.12) is responsible for issuing periodic queries against public FRC Web services that provide access to the most current policy information. These queries help confirm that current policies used by Alleywood are still valid or have changed.

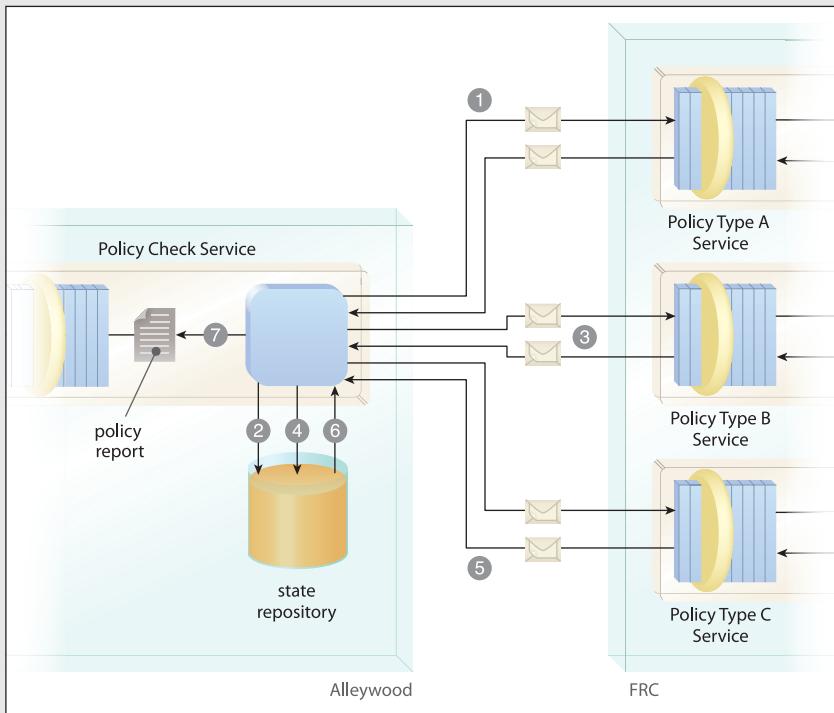


Figure 9.12

The Policy Check service issues a query against the first FRC service (1) and then writes the results to the state repository (2) before requesting data from the next FRC service (3). That information is then also written to the state repository (4), and after the Policy Check service retrieves the last batch of data from the third FRC service (5), it retrieves the data from the state repository (6) and assembles the requested policy report (7).

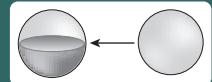
The FRC exposes three separate Web services that provide report-style data for different policy types. Alleywood, being a larger-sized company, is required to remain in compliance with all three types of policies. This service therefore needs to query each FRC service before it can produce a consolidated report.

Query times can vary, depending on concurrent usage of the FRC services and the databases they access. Sometimes it can take minutes to receive a response, and other times the response times out or fails altogether. The initial version of the Policy Check service had many problems due to these irregular access conditions. It became one of the most unreliable parts of the Alleywood service inventory and consumed unusually high amounts of memory.

As a result, the Policy Check service is refactored to write each batch of data it receives to a state repository. Even if access to an FRC service fails, the data collected so far is preserved in this database, while the Policy Check service retries its access. It then continues to wait until it has received all the information it needs, at which point it retrieves all of the data back from the state repository and merges it into the requested report.

Stateful Services

How can service state data be persisted and managed without consuming service runtime resources?



Problem	State data associated with a particular service activity can impose a great deal of runtime state management responsibility upon service compositions, thereby reducing their scalability.
Solution	State data is managed and stored by intentionally stateful utility services.
Application	Stateful utility services provide in-memory state data storage and/or can maintain service activity context data.
Impacts	If not properly implemented, stateful utility services can become a performance bottleneck.
Principles	Service Statelessness
Architecture	Inventory, Service

Table 9.4

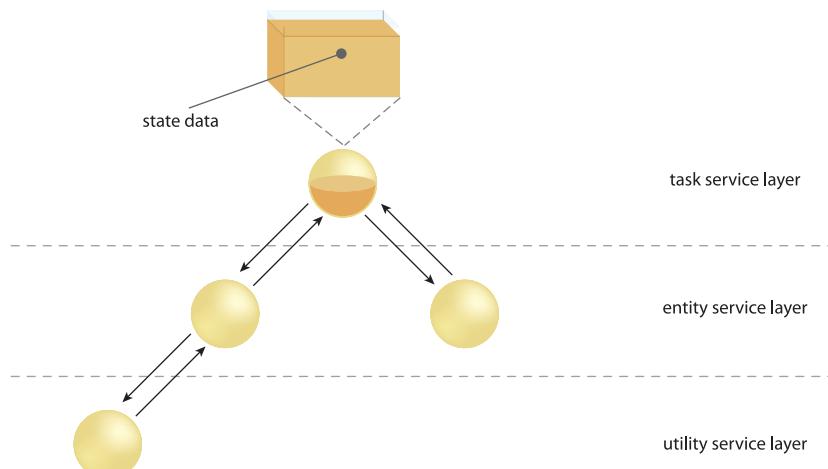
Profile summary for the Stateful Services pattern.

Problem

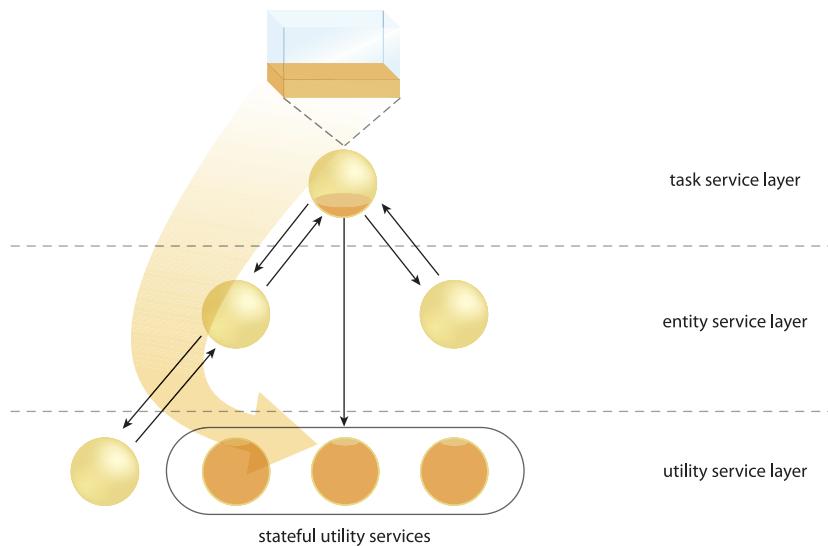
The coordination of large service activities requires the constant management of state data. Placing the burden of retaining and processing this data upon business services increases their individual memory consumption as well as the duration for which they are required to remain stateful (Figure 9.13).

Solution

Intentionally stateful utility services are created to provide regular state deferral and storage functions and/or to provide runtime coordination functions to manage the flow and participation of service activities. This alleviates the need for any one business service from having to retain or manage state data for extended periods (Figure 9.14).

**Figure 9.13**

The task controller service of a modest composition is required to retain and manage all of the service activity's state data until the activity is completed.

**Figure 9.14**

With the use of stateful utility services, state management responsibilities are deferred.

Application

This pattern is commonly applied in two ways:

- The stateful utility services provide state management deferral functions that are explicitly used by other services as required.
- The stateful utility services are part of a service activity management framework (such as WS-Coordination) within which they act as runtime activity coordinators.

Either way, what distinguishes services dedicated to state management is that they are deliberately stateful parts of the enterprise. Therefore, these specialized services intentionally violate the Service Statelessness principle so as to support its application in other services.

NOTE

When stateful utility services act as coordinators during the execution of a service activity, the type of state data they process is commonly referred to as *context data* in that it represents information pertaining to the context of the current service activity.

Impacts

In high concurrency situations, stateful utility services can be required to manage numerous service activities and activity instances at the same time. If they are not supported by the proper infrastructure, the overall performance and scalability of the service inventory as a whole can be compromised, thereby undermining their purpose.

Also the use of stateful utility services adds more “moving parts” to a given service composition, thereby increasing its complexity.

Relationships

This pattern establishes a specialized variation of the utility service and is therefore related to Utility Abstraction (168). Some implementations may still require a state management database behind the scenes, leading to the need to also apply State Repository (242) and the option to utilize State Messaging (557) to temporarily off-load state data is also possible. Additionally, both State Repository (242) and State Messaging (557) represent viable alternatives to Stateful Services altogether.

When stateful utility services exist as Web services, Service Messaging (553) is required for basic communication, and Messaging Metadata (538) provides a means of supplementing state data deliveries with additional activity details. As further explored in the next pattern description, Stateful Services also relates closely to Service Grid (254).

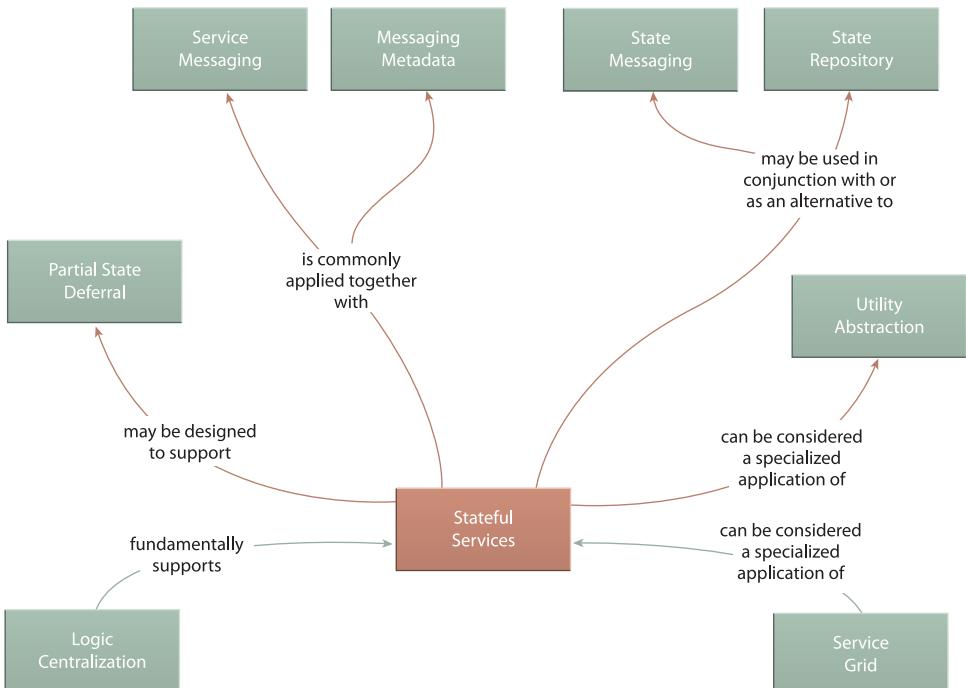


Figure 9.15

Stateful Services results in the creation of utility services that centralize state and activity management, primarily via messaging.

CASE STUDY EXAMPLE

The transaction framework described in the case study example for Canonical Resources (237) established WS-Coordination as a standardized context management system intended to facilitate all ACID-style transactions within the FRC service inventory.

Under the covers this framework is comprised of a set of stateful utility services that are pre-defined as part of the WS-Coordination specification. These services require that regular services participating in a transaction first register for the transaction and then communicate to them the status of their involvement. Once a service has completed its participation, it de-registers itself from the transaction.

The rules by which the stateful WS-Coordination services manage transactions are defined separately in the WS-AtomicTransaction specification. For example, a voting mechanism is introduced whereby participating services are polled as to whether their contribution to a given transaction was successful or not. Services can respond with “Commit” or “Abort” messages that indicate their status. If just one “Abort” message is received (or if one vote is missing from the registered services), then the transaction is in fact aborted, and a “Rollback” message is sent to all participants.

Together, WS-Coordination and WS-AtomicTransaction provide an industry-standard transaction management framework for the FRC that manages service activity data (referred to as context information) on behalf of other custom services. This alleviates custom FRC services from having to provide some of the logic required to coordinate service activity-specific details.

However, after working with this framework, it is soon discovered that there is an additional opportunity to delegate state management-related processing. Specifically, FRC architects notice redundant logic creeping into a number of entity services required to work with a set of code lists common to the forestry industry. Often these code lists need to be placed into memory and then repeatedly accessed as transactions are carried out.

FRC architects would like to see this type of state data managed separately by stateful utility services. Because the WS-Coordination system services are pre-defined, architects are not comfortable augmenting them to incorporate this new functionality. Instead, they opt to create a custom Code service that will be used to complement the WS-Coordination framework by allowing all services participating in transactions to read and write code lists (Figure 9.16).

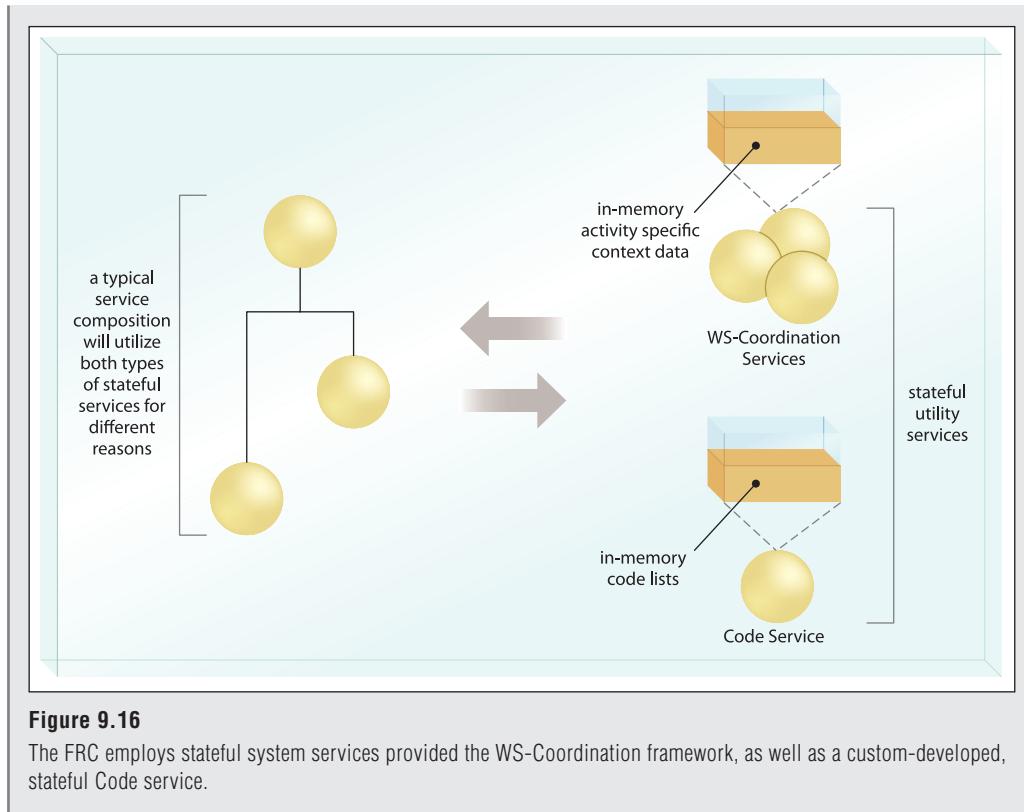


Figure 9.16

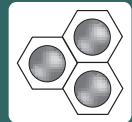
The FRC employs stateful system services provided the WS-Coordination framework, as well as a custom-developed, stateful Code service.

NOTE

The preceding case study example used the WS-Coordination framework as an example of a framework that supports the application of this pattern. The actual mechanics behind cross-service transactions are further explained in the pattern description for Atomic Service Transaction (623).

Service Grid

By David Chappell



How can deferred service state data be scaled and kept fault-tolerant?

Problem	State data deferred via State Repository or Stateful Services can be subject to performance bottlenecks and failure, especially when exposed to high-usage volumes.
Solution	State data is deferred to a collection of stateful system services that form a grid that provides high scalability and fault tolerance through memory replication and redundancy and supporting infrastructure.
Application	Grid technology is introduced into the enterprise or inventory architecture.
Impacts	This pattern can require a significant infrastructure upgrade and can correspondingly increase governance burden.
Principles	Service Statelessness
Architecture	Enterprise, Inventory, Service

Table 9.5

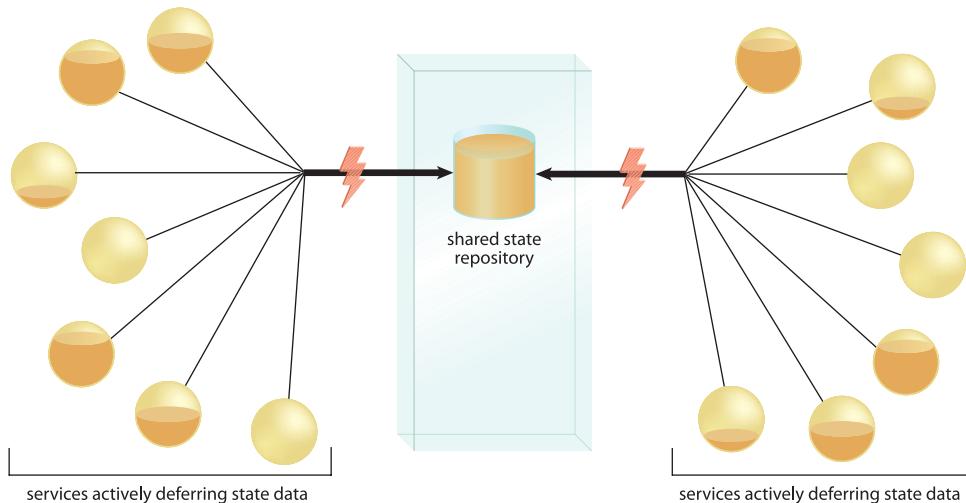
Profile summary for the Service Grid pattern

Problem

Conventional state deferral mechanisms have thresholds that can impede the usage potential of services.

For example:

- When services defer state data to a central database, as per State Repository (242), it can result in performance bottlenecks relative to the extent that the repository is shared and the available resources of the underlying infrastructure. Furthermore, a state database can become a single point of failure for all services that rely on it (Figure 9.17).
- When services defer state data to utility services, as per Stateful Services (248), failover concerns are even greater than with State Repository (242) because the state data is kept in memory and may not be recoverable after a failure condition. Additionally, stateful utility services may become performance bottlenecks due to an absence of built-in load balancing functionality.

**Figure 9.17**

A central state repository can raise performance and reliability concerns when it is subject to high concurrent usage.

In some platforms, State Repository (242) and Stateful Services (248) can be supported by infrastructure extensions that provide failover. However, these extensions are often based on “failure and restart” approaches that involve a transaction manager-like rollback and recovery. While this provides some level of fault tolerance, it will typically result in loss of data, runtime disruption and exceptions, and may further require manual intervention by humans.

Solution

Deferred service state data is persisted and stored by a dedicated collection of grid services—stateful services which are part of a services-based grid platform and act as an extension of the infrastructure.

Within this platform, multiple, redundant instances of the grid services are constantly available and remain consistently synchronized. This allows each grid service to provide its own individual memory cache that is replicated across multiple redundant instances that reside on and are load balanced across different server machines (Figure 9.18). Additional grid service instances can be further spawned, as required.

The resulting environment can establish high scalability and fault tolerance of deferred state data throughout an entire service inventory and even across multiple inventories.

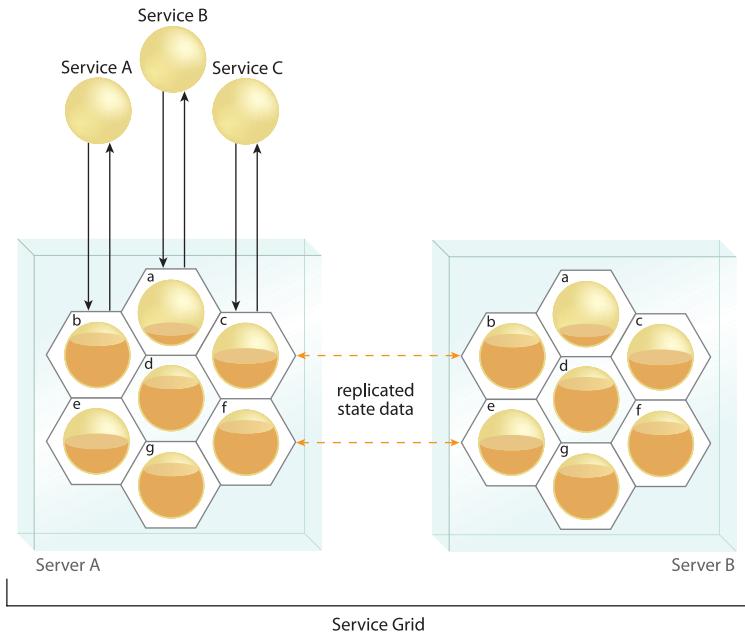


Figure 9.18

A service grid establishes replicated instances of stateful grid services across different server machines, resulting in increased scalability and reliability of state data. (A grid service is represented by the standard service symbol enclosed in a honeycomb cell.)

Application

How Service Grid is actually implemented can vary, depending on the specific platform or vendor product that is chosen. A common process is for a custom service to pass state data to a grid service, which then responds with a unique identifier (called a *state key*) that represents the body of state data. The service receives and holds onto the state key while it remains active, and can then use this key to access and retrieve some or all of the previously deferred state data.

NOTE

The notion of a state key also forms the basis of a separate pattern that allows the same body of state data to be shared across multiple services and service compositions. This and other specialized patterns associated with Service Grid are being published at SOAPatterns.org and will further be documented in a separate book by David Chappell.

Behind the scenes, the inner mechanics of the service grid ensure that whatever state data is received is constantly duplicated via distributed, redundant grid service instances. If the custom service makes a change to the state data or retrieves portions of it, these events are replicated to the corresponding redundant grid service instances so that they remain synchronized.

Should a grid instance fail, any one of its counterparts assumes its place and continues to make the state data available to the original custom service. Intelligent load-balancing functionality may be present to direct deferral or retrieval requests from the custom service to the grid service instance residing on the physical server that is being used the least at that point in time. Furthermore, advanced grid computing extensions can be added to offload the execution of service logic into the service grid in order to reduce network data serialization latency between custom and grid services.

Throughout all of this, regular custom services that interact with grid services are shielded from the inner workings of the service grid platform and may simply view grid services as generic stateful utility services. A service grid implementation can include or be further extended with State Repository (242) for long-term state storage requirements.

This pattern is especially effective in large-scale service inventories or across multiple inventories because of its horizontal scalability potential. It is not uncommon for service grid implementations to be comprised of dozens or hundreds of servers. The constant availability of the state deferral mechanism provided by the grid services reduces the resource impact on regular custom services, thereby increasing their scalability as well. When broadly utilized, this load sharing dynamic can establish a service grid as a prevalent and intrinsic part of the overall service-oriented enterprise.

NOTE

The actual type of interface or technical contract exposed by grid services can vary, depending on the grid platform.

Impacts

The need to add multiple physical servers, coupled with product license costs and additional required infrastructure extensions can make the adoption of Service Grid costly. It may be further desirable for a service grid to be isolated on its own high-speed network in order to accommodate the constant cross-server synchronization that needs to occur.

As a result of the required expansion of infrastructure, grid-based environments will naturally increase the governance burden of one or more service inventory architectures, resulting in on-going operational effort and costs.

Relationships

The application of Service Grid essentially results in the application of Stateful Services (248), but State Repository (242) can also become part of a grid platform, depending on its configuration. Partial State Deferral (356) is generally supported by grid services, that may further require the use of Messaging Metadata (538) to exchange state keys.

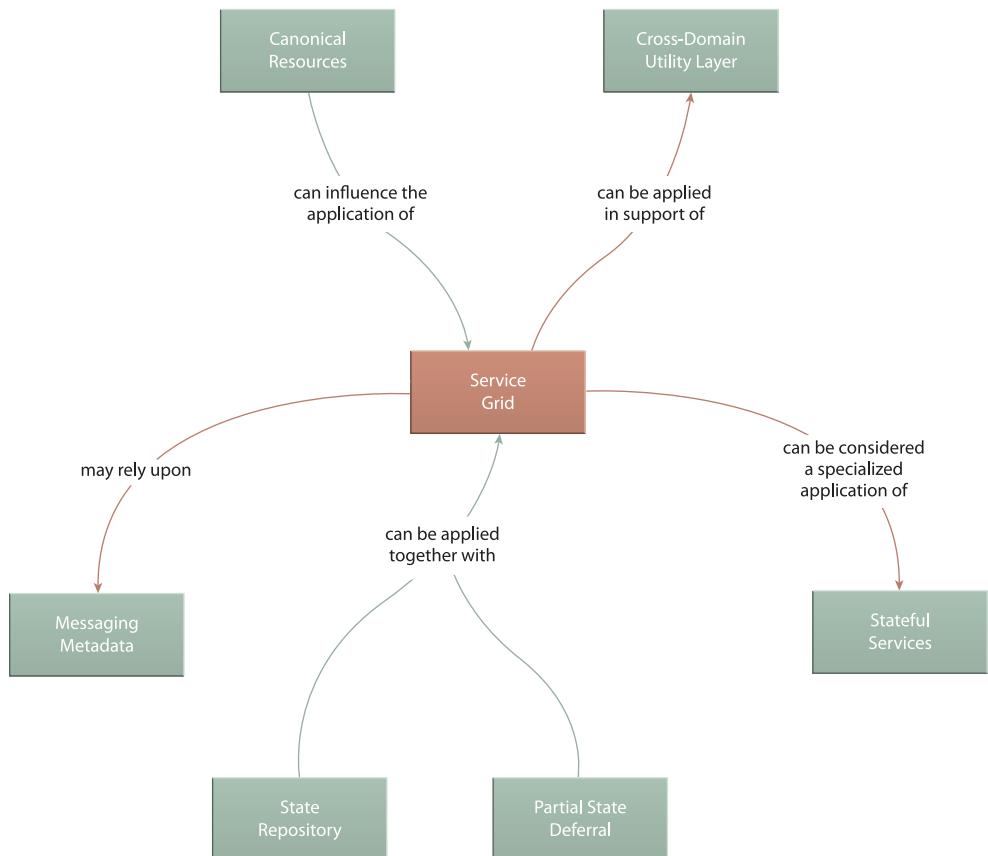


Figure 9.19

Service Grid naturally relates to other state management patterns and others associated with its inner mechanics.

When positioned as an enterprise-level resource, Service Grid can establish infrastructure that can be leveraged by multiple service inventories. Because of the utility-centric nature of grid services, this can effectively enable or extend the application of Cross-Domain Utility Layer (267).

CASE STUDY EXAMPLE

In the case study example for Stateful Services (248), the FRC proceeded with an architecture whereby they built a Code service to act as a state management resource. Soon after this went into production, many additional requirements for various specialized state deferral scenarios emerge.

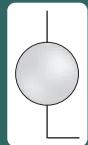
Additionally, FRC architects witness how the individually deployed Code service becomes increasingly popular. After warnings from systems administrators that the service is nearing its concurrent usage threshold, a spike in usage causes it to exhaust available memory, triggering runtime exceptions and ultimately resulting in a system failure that cascades across the various service compositions that were using the service.

This experience convinces architects to immediately begin investigating options to increase scalability and reliability of this and other state management services they were planning to deliver. In the short-term, they apply Redundant Implementation (345) to provide some relief, but their attention soon turns toward Service Grid.

Based on current usage statistics, they find it difficult to warrant the investment of a full-blown grid computing platform. However, a subsequent study of upcoming projects and a review of their service inventory blueprint convinces them that this will eventually be needed and that a service grid platform should be established soon so that it can evolve with the rest of the service inventory architecture. This will allow the planned services and compositions to incorporate the grid services and the use of state keys right away as part of their initial design. The savings in avoided refactoring costs alone reassures the architects of their decision to proceed with Service Grid.

Inventory Endpoint

How can a service inventory be shielded from external access while still offering service capabilities to external consumers?



Problem	A group of services delivered for a specific inventory may provide capabilities that are useful to services outside of that inventory. However, for security and governance reasons, it may not be desirable to expose all services or all service capabilities to external consumers.
Solution	Abstract the relevant capabilities into an endpoint service that acts as the official inventory entry point dedicated to a specific set of external consumers.
Application	The endpoint service can expose a contract with the same capabilities as its underlying services, but augmented with policies or other characteristics to accommodate external consumer interaction requirements.
Impacts	Endpoint services can increase the governance freedom of underlying services but can also increase governance effort by introducing redundant service logic and contracts into an inventory.
Principles	Standardized Service Contract, Service Loose Coupling, Service Abstraction
Architecture	Inventory

Table 9.6

Profile summary for the Inventory Endpoint pattern.

Problem

As described in Chapter 4, a service inventory represents a collection of independently standardized and governed services. When opportunities arise for services to share their capabilities with service consumers that reside outside of the inventory (whether they are consumers within the same organization but part of a different inventory or consumers external to the organization itself), interoperability, privacy, and security-related concerns often arise, making the option of simply exposing internal inventory services to external consumers less than desirable (Figure 9.20).

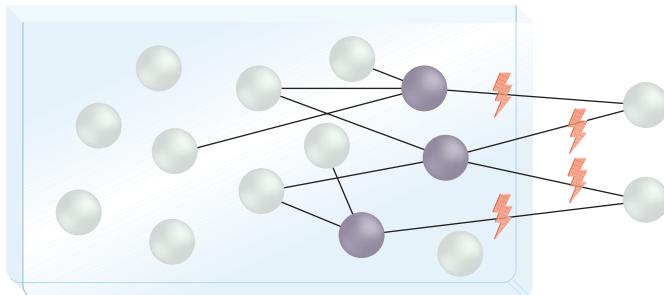


Figure 9.20

External service consumers accessing native inventory services can impose runtime demands and introduce new risks.

Solution

A special type of intermediary service is positioned as the official service inventory entry point for consumers external to the inventory that need to access native services within the inventory (Figure 9.21). This endpoint service can be configured to accommodate consumer interaction preferences and can further contain broker and mediation logic to help facilitate communication with internal inventory services.

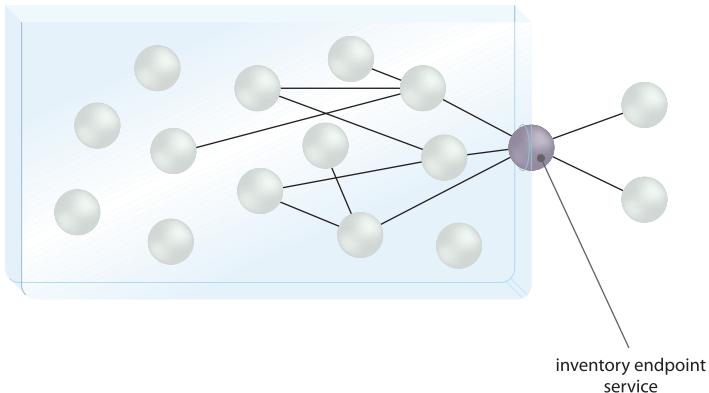


Figure 9.21

A new service introduced to facilitate external consumer requirements can ensure that other native inventory services are not affected.

Application

By abstracting capabilities from a collection of services into a single contract, services positioned as endpoints for an inventory offer several benefits, including:

- Increased governance freedom for the underlying services, as they can be changed and extended without affecting the endpoint service contract. Even if underlying service functionality needs to be altered, logic could be introduced into the endpoint service to accommodate for the changes so that external consumers remain unaffected and unaware.
- The endpoint service contract can be fully customized to accommodate the external consumer programs. This allows for the addition of data and security constraints, policy assertions and alternatives, and even the support of additional transport protocols unique to the consumer interaction requirements. By abstracting these implementation requirements into a single service, underlying inventory services are not required to change.
- A separate endpoint service can be created for each group of external consumers. This allows the aforementioned customization to be specific to a range of consumer types. For example, one endpoint service can be created for consumers from a different domain inventory, and a separate endpoint service can be positioned for consumer programs residing outside of the organization itself.
- Beyond providing alternative contract representation for inventory services, an endpoint service can also provide Protocol Bridging (687) for consumers that use disparate protocols or data exchange technologies.

Endpoint services are typically single-purpose with non-agnostic functional contexts and are therefore generally classified as task services. Some organizations, however, prefer to consider the endpoint service as its own service model, especially since endpoint services may be required to encapsulate inventory-specific task services.

Although they are often delivered and owned by the custodian of the inventory for which they act as endpoints, they are not always considered members of that inventory because they are required to conform to different design standards and are not made available for native compositions. Endpoint services are often literally maintained at the periphery of inventory boundaries. Therefore, the first step to working with endpoint services is to establish an effective ownership structure that will allow these services to evolve with both their underlying inventories and their consumers.

For endpoint services created to interact with consumers from external organizations, special implementation requirements are almost always needed. These can include the need for deployment within a DMZ on an isolated server and various infrastructure extensions associated with security and sometimes scalability.

The core service logic for an endpoint service is generally comparable to logic shaped by Service Façade (333) in that it is mostly comprised of routines that relay data requests and responses to and from the external consumers and the underlying inventory services. However, when endpoint services are required to provide new policies or enforce new constraints, additional logic is needed. Furthermore, endpoint services are commonly relied upon to act as brokers by carrying out Data Model Transformation (671), Data Format Transformation (681), and even Protocol Bridging (687).

Impacts

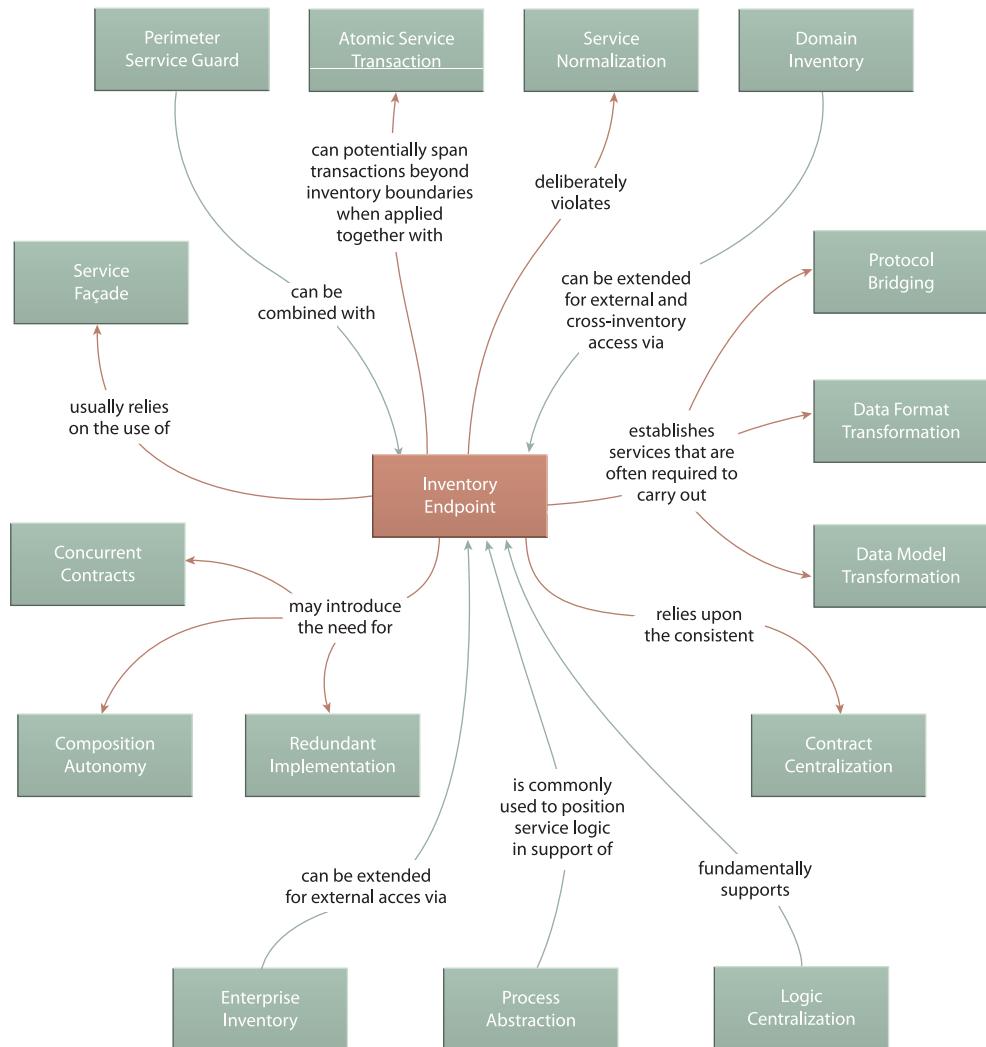
While Inventory Endpoint increases the freedom with which inventory services can be evolved and governed over time, they do result in the introduction of new services and service contracts that will need to be maintained as an addition to the service inventory itself. This governance responsibility and the associated ownership issues that need to be addressed can introduce a significant amount of cost and effort because of the on-going maintenance required to keep them in sync with internal service and external consumer requirements. This pattern may even lead to the need for a new IT group altogether (especially if multiple endpoint services are produced).

Relationships

The use of Inventory Endpoint raises both contract design and architectural issues, which therefore relates this pattern to service design patterns, such as Service Façade (333) and Concurrent Contracts (421), as well as implementation-related patterns like Composition Autonomy (616) and Redundant Implementation (345).

In fact, this pattern can sometimes appear as a specialized variation of Concurrent Contracts (421) in that it introduces the need to establish new services that functionally overlap with existing ones (and therefore also violates Service Normalization (131) to an extent).

As shown by the relationships to the three patterns that comprise Service Broker (707) on the right side of Figure 9.22, one of the most common responsibilities of the inventory endpoint service is to overcome the communication disparity between inventory services and external consumers. This is simply because consumers outside of the inventory are generally subject to different design standards and conventions.

**Figure 9.22**

Inventory Endpoint provides a specialized design solution that touches on a range of design issues.

NOTE

The application of Enterprise Service Bus (704) will also often naturally apply Inventory Endpoint by establishing external endpoints that encapsulate broker and mediation logic. The distinction with this pattern is that the endpoint is specific to a service inventory.

CASE STUDY EXAMPLE

A new Tri-Fold service composition is assembled to automate the recently modeled Plant Supply business process. This is a complex composition involving eight services and a great deal of activity management. To further complicate the design, the Plant Supply task service is required to access the following three services that reside within the Alleywood service inventory:

- a Trucks service responsible for processing information related to the delivery of materials to the plants
- a Load service that provides functionality pertaining to “in transport” materials being delivered
- a Mills service that represents the origin of delivered loads

The initial composition design has the Plant Supply task service invoking each of the three Alleywood services individually, thereby being subject to remote access performance challenges and data model transformation requirements, in addition to further disparity in how security and activity meta data is represented (Figure 9.23).

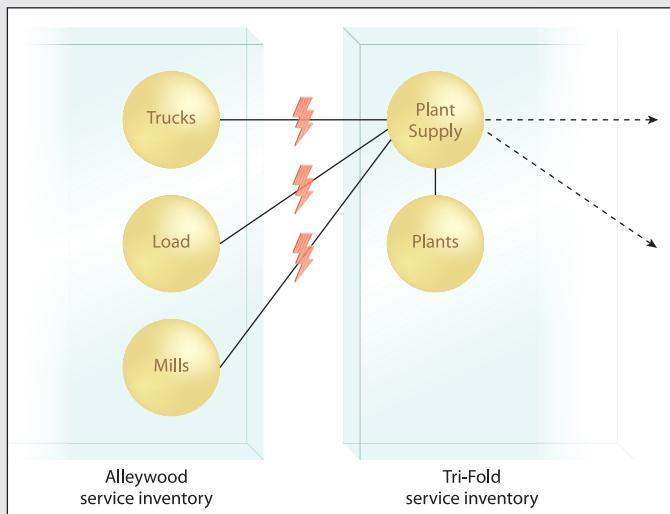


Figure 9.23

Repeated cross-domain access by the Tri-Fold Plant Supply task service compounds the impact of the disparity between the independent service inventories.

Once performance calculations are added up and after the design complexity of this proposed composition is mapped out, the approach is rejected. Instead, by applying Inventory Endpoint, a new service is introduced into the Alleywood inventory called the Plant Supply Endpoint service.

This service establishes a contract custom designed for the Tri-Fold Plant Supply service so that no external transformation is required (Figure 9.24). Internally, this service performs all necessary conversion and also encapsulates the required composition logic to interact with the Alleywood Trucks, Load, and Mills services.

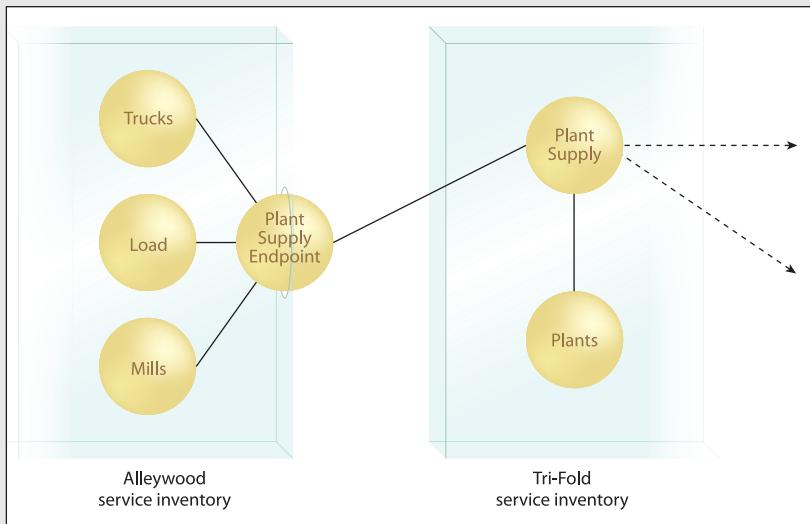
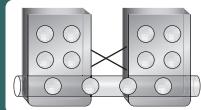


Figure 9.24

The new Plant Supply endpoint service eliminates unnecessary remote communication and carries out all required composition and transformation logic.

Cross-Domain Utility Layer

How can redundant utility logic be avoided across domain service inventories?



Problem	While domain service inventories may be required for independent business governance, they can impose unnecessary redundancy within utility service layers.
Solution	A common utility service layer can be established, spanning two or more domain service inventories.
Application	A common set of utility services needs to be defined and standardized in coordination with service inventory owners.
Impacts	Increased effort is required to coordinate and govern a cross-inventory utility service layer.
Principles	Service Reusability, Service Composability
Architecture	Enterprise, Inventory

Table 9.7

Profile summary for the Cross-Domain Utility Layer pattern.

Problem

The primary reason for enterprises to proceed with multiple domain service inventories is to allow for the governance of individual inventories by separate groups that represent the respective domains. More often than not, these inventories are associated with organizational business domains, and the governance issues pertain to the design and evolution of business service layers. The rationale is to tolerate the use of different standards and increased redundancy across business service layers within domains for the benefit of achieving manageable SOA adoption and governance.

However, the utility layers within these domains have no ties to business models, and often the corresponding utility services encapsulate enterprise resources that are common to all domains. As a result, some utility logic created for one domain will tend to be functionally similar (or even identical) to others. The resulting redundancy and design disparity within multiple utility service layers (across different inventories) is therefore wasteful and unnecessary (Figure 9.25).

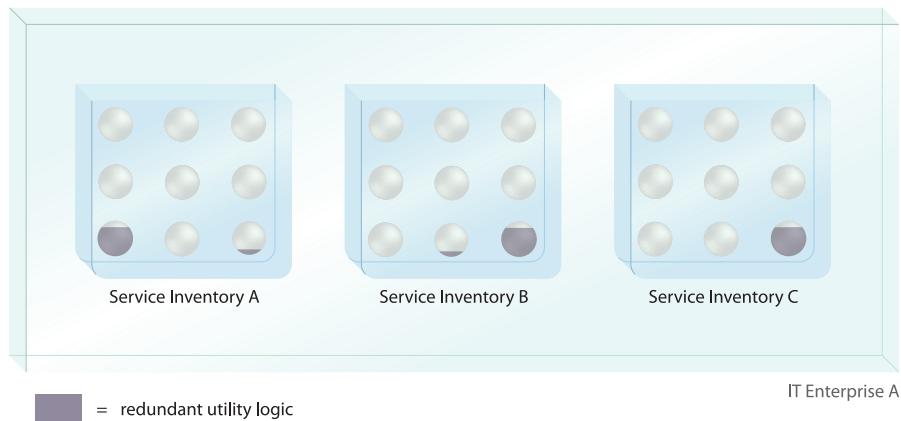


Figure 9.25

By having to duplicate functionality across domain service inventories, more utility logic and services are created than are actually required.

Solution

A common utility service layer is positioned for use by multiple domain service inventories, establishing a centralized collection of normalized (non-redundant) utility services accessible to and reusable by services across domains (Figure 9.26).

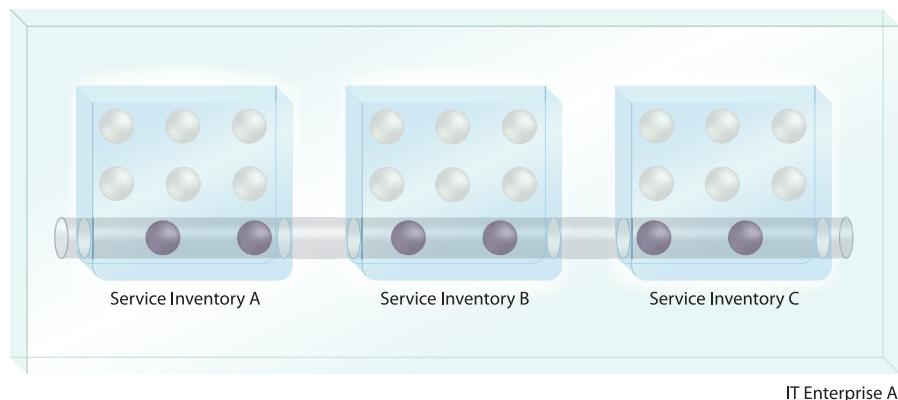


Figure 9.26

A cross-domain utility service layer establishes a set of common services that address broad, cross-cutting concerns. Notice how a smaller quantity of utility services is required (compared to Figure 9.25) due to reduced redundancy.

Application

It is recommended that in addition to design standards that require domains to use utility services, standard processes also exist across domains to allow for the identification and reuse of cross-domain utility services. This service layer is very much a part of the enterprise architecture and should therefore be established prior to domain service inventory definition.

Note that a cross-domain utility service layer does not need to replace a domain inventory's utility layer in its entirety. Domain-specific utility services can be defined as required and then further complemented by cross-domain utility services (Figure 9.27).

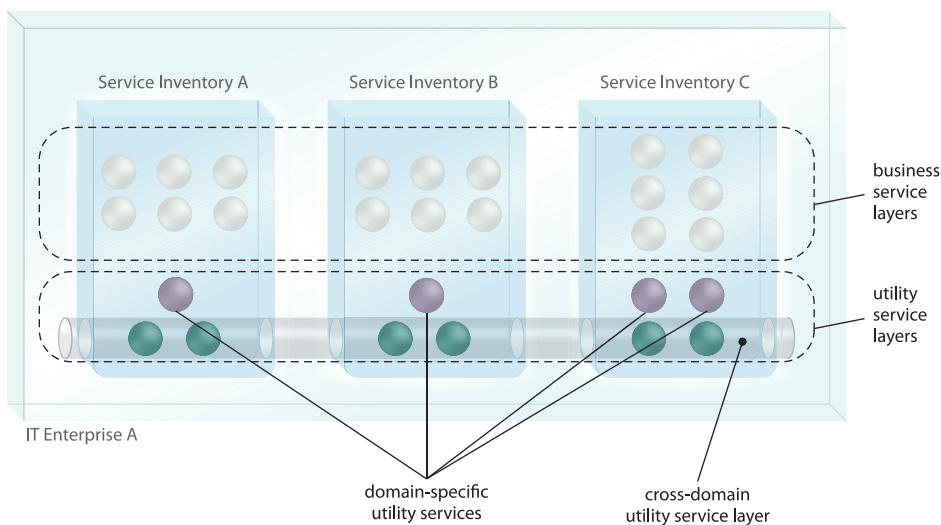


Figure 9.27

An enterprise architecture comprised of three inventories that share a cross-domain utility service layer but also allow for domain-specific utility services to be created.

Impacts

One of the reasons to create domain service inventories is to allow for each domain to evolve independently, which is a more manageable approach for some organizations. Requiring that all inventories use the same common set of utility services reduces this independence somewhat.

It furthermore complicates the overall governance processes that need to be in place; instead of domain-specific groups that own and maintain domain services, there may now

need to be an enterprise governance group that owns the cross-domain utility service layer and ensures that these services are properly utilized within each domain.

Note also that if service inventory domains are based on geographical boundaries, or if domains consist of vastly disparate technical environments, the governance logistics for applying this pattern can prove difficult.

Relationships

Cross-Domain Utility Layer changes the complexion of a service-oriented enterprise by impacting multiple domain inventory architectures and therefore has naturally close relationships with Domain Inventory (123), Utility Abstraction (168), and Agnostic Context (312), while also providing an opportunity to establish broad baseline interoperability in support of Canonical Protocol (150).

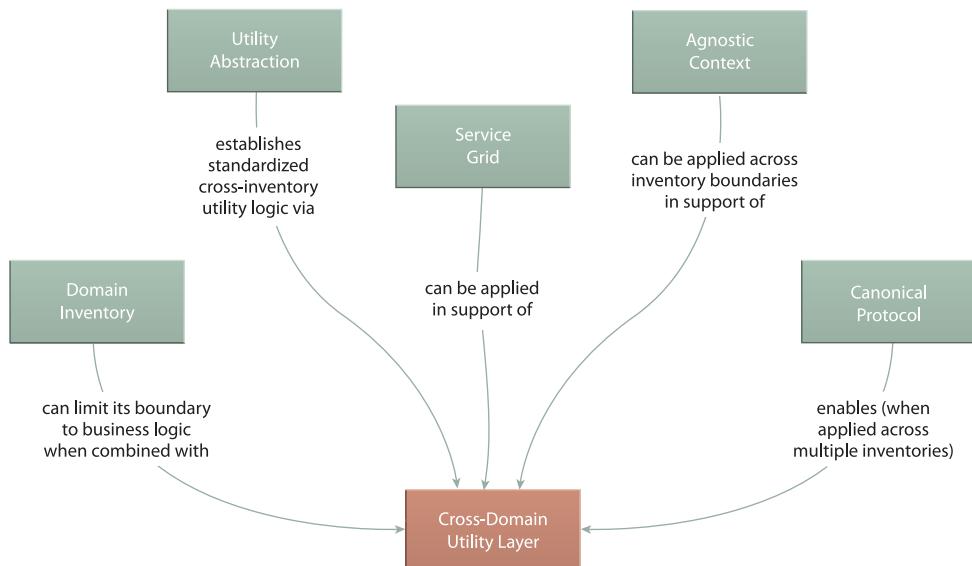


Figure 9.28

Cross-Domain Utility Layer increases the reach of shared utility services in support of increased recomposition of utility capabilities.

CASE STUDY EXAMPLE

When both Alleywood and Tri-Fold service inventory blueprints are completed, they are reviewed by the McPherson Enterprise Group. Although each is comprised of well-defined, normalized collections of services, it is evident that there is significant

functional redundancy, especially within the utility service layers. In fact, the utility logic defined is almost identical, primarily due to these services being positioned for access to shared resources.

The reasons behind splitting the original enterprise service inventory into separate physical domains were primarily rooted within resource and governance issues associated with business concerns. However, there is no real objection to applying this pattern to establish an enterprise-wide utility service layer. Although the initial layer is comprised of all utility services either inventory needs (Figure 9.29), Alleywood and Tri-Fold will be able to create inventory-specific utility services as well.

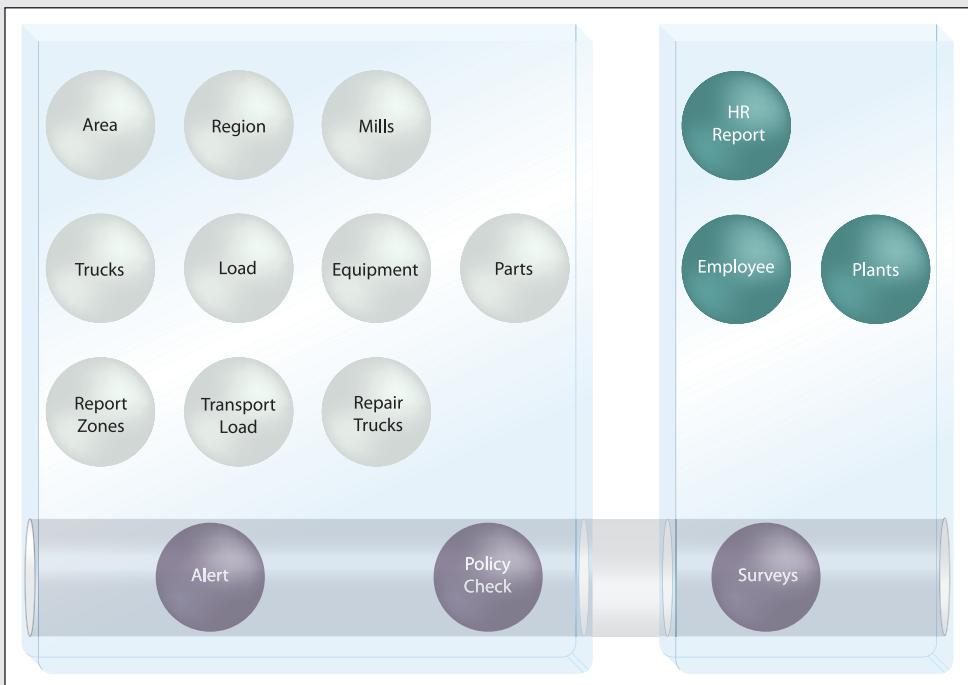
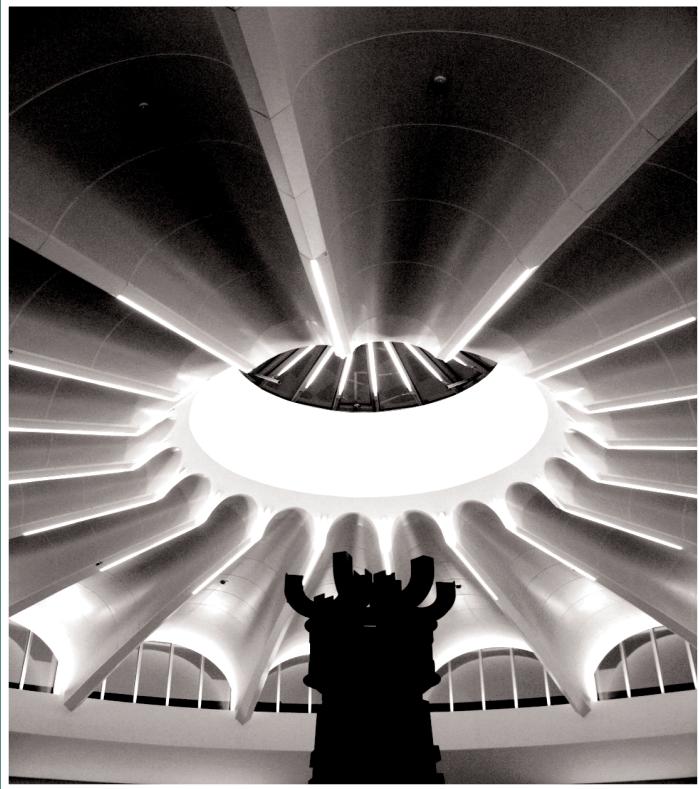


Figure 9.29

A cross-domain utility service layer that effectively replaces both previously defined inventory-specific utility service layers.

This page intentionally left blank

Chapter 10



Inventory Governance Patterns

Canonical Expression

Metadata Centralization

Canonical Versioning

When first designing a service inventory, there are steps that can be taken to ensure that the eventual effort and impact of having to govern the inventory is reduced. This chapter provides a set of patterns that supply some fundamental design-time solutions specifically with the inventory's post-implementation evolution in mind.

Canonical Expression (275) refines the service contract in support of increased discoverability, which goes hand-in-hand with Metadata Centralization (280), a pattern that essentially establishes a service registry for the discovery of service contracts. These patterns are further complemented by Canonical Versioning (286), which requires the use of a consistent, inventory-wide versioning strategy.

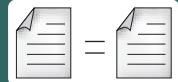
All of these patterns are considered fundamental to inventory governance in that they support and are influenced by the Service Discoverability principle, which actually shapes service meta information in such a manner that it can be effectively discovered and interpreted.

NOTE

The governance patterns in this chapter focus on fundamental technical and design-related governance issues only. The upcoming title *SOA Governance* as part of this book series will provide a collection of additional technical and organizational best practices and patterns.

Canonical Expression

How can service contracts be consistently understood and interpreted?



Problem	Service contracts may express similar capabilities in different ways, leading to inconsistency and risking misinterpretation.
Solution	Service contracts are standardized using naming conventions.
Application	Naming conventions are applied to service contracts as part of formal analysis and design processes.
Impacts	The use of global naming conventions introduces enterprise-wide standards that need to be consistently used and enforced.
Principles	Standardized Service Contract, Service Discoverability
Architecture	Enterprise, Inventory, Service

Table 10.1

Profile summary for the Canonical Expression pattern.

Problem

Service contracts delivered or extended by different projects and at different times are naturally shaped by the architects and developers that work with them. The manner in which the service context and the service's individual capabilities are defined and expressed through the contract syntax can therefore vary. Some may use descriptive and verbose conventions, while others may use terse and technical formats. Furthermore, the actual terms used to express common or similar capabilities may also vary.

Because services are positioned as enterprise resources, it is fully expected that other project teams will need to discover and interpret the contract in order to understand how the service can be used. Inconsistencies in how technical service contracts are expressed undermine these efforts by introducing a constant risk of misinterpretation (on a technical level). The proliferation of these inconsistencies furthermore places a convoluted face on a service inventory, increasing the effort to effectively navigate various contracts to study possible composition design options.

Solution

Standardized naming conventions can be applied to the delivery of all service contracts so as to ensure the consistent expression of service contexts and capabilities (Figure 10.1).

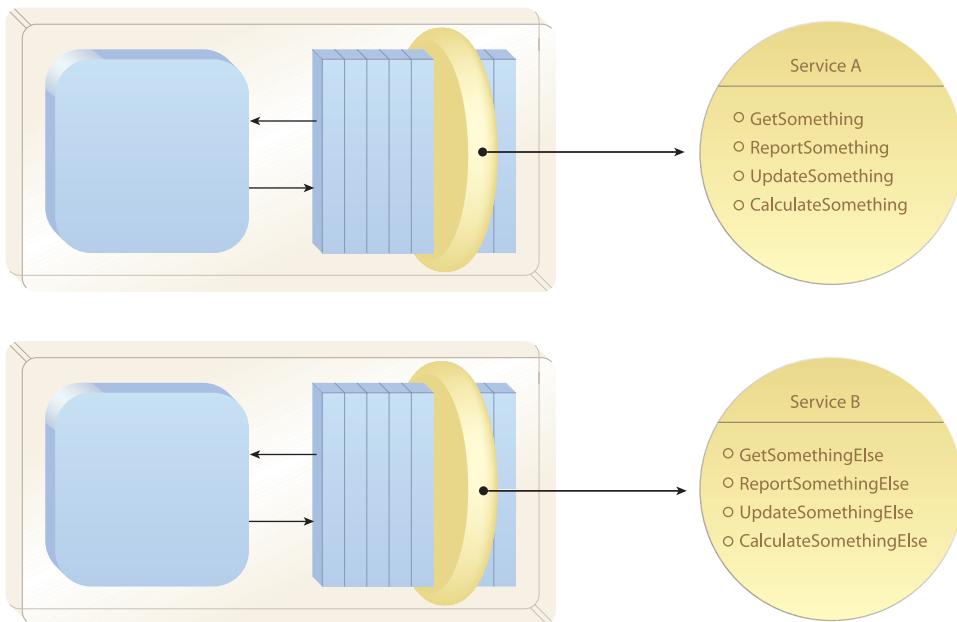


Figure 10.1

The expression of service contracts is aligned across services.

Application

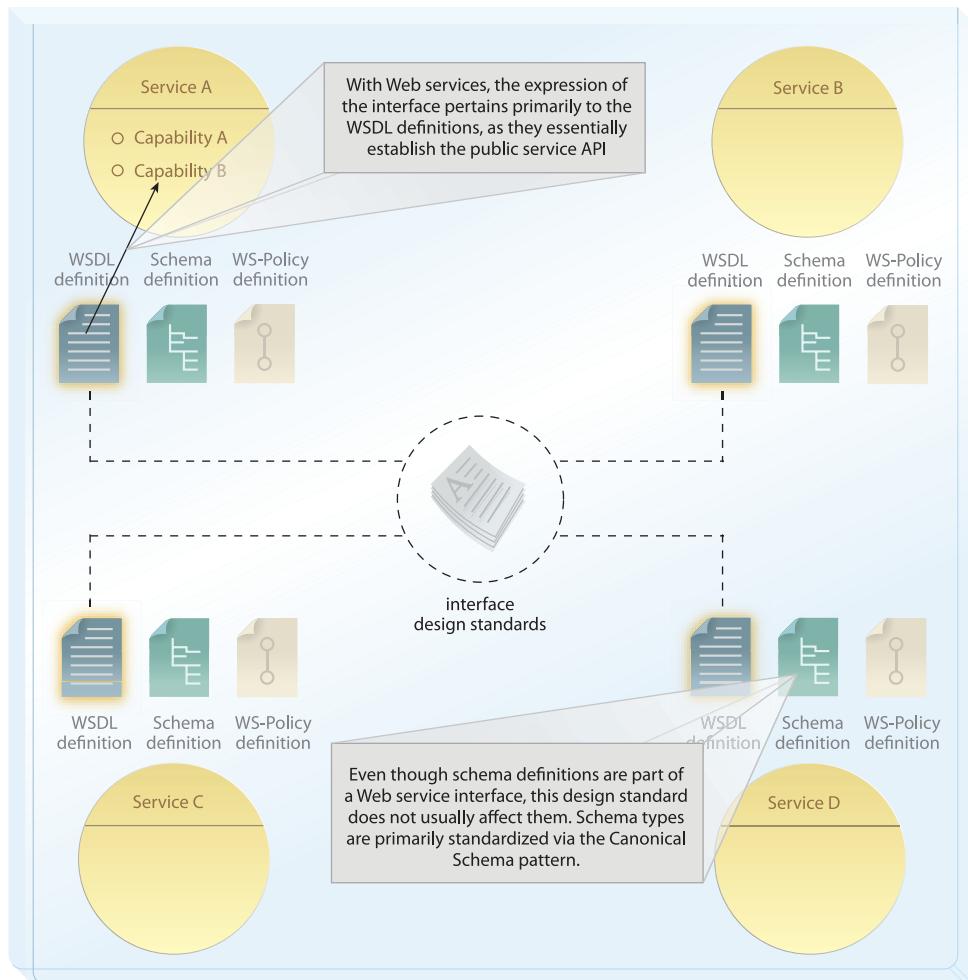
A set of naming and functional expression conventions needs to be established as formal design standards. The realization of consistent contract design is then attained via the disciplined use of these conventions within common analysis and design processes.

An example of a standard associated with contract expression is the CRUD (create, read, update, delete) convention traditionally used to outfit components with a predictable set of methods. Entity services in particular often require these types of data processing functions, and using standardized verbs to express them supports the application of this pattern.

With Web services in particular, this pattern will tend to impact the design of WSDL definitions, as illustrated in Figure 10.2.

NOTE

This pattern can be applied regardless of whether the service contract is decoupled.

**Figure 10.2**

The WSDL definitions of the four services are affected by Canonical Expression.

Impacts

The relevance of Canonical Expression may at first appear trivial. However, when building a collection of services, especially within larger enterprise environments, a consistent functional expression significantly reduces tangible risk factors.

The primary requirement to successfully applying this pattern is the incorporation and enforcement of the required design standards. If a formal design process has already been

established in support of Decoupled Contract (401) and Canonical Schema (158), then the effort to include a step dedicated to Canonical Expression is usually minor.

Note also that unlike Canonical Schema (158), which often must be limited to domain service inventories due to its governance impact, this pattern can more easily be positioned as an enterprise-wide standard. This benefits the enterprise as a whole as consistent expression is established across all domains.

Relationships

The naming conventions introduced by Canonical Expression influence how several other patterns are applied (as listed at the top of Figure 10.3). This pattern fundamentally supports the goals of Contract Centralization (409) and Metadata Centralization (280) by enhancing the intuitiveness of service identification and reuse.

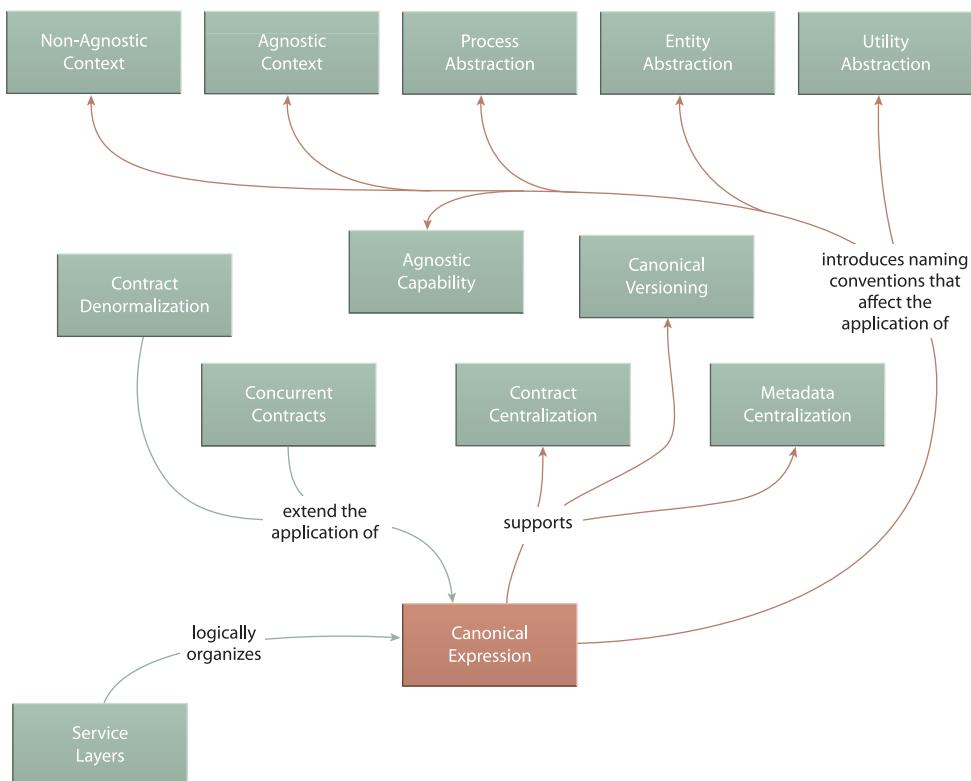


Figure 10.3

Canonical Expression keeps the external expression of service contracts consistent, thereby affecting contract and context-related patterns.

CASE STUDY EXAMPLE

An early pilot version of the Inventory Processing service has been used for testing purposes. It consists of a Web service that was auto-generated using a development tool that derived the Web service contract from component class interfaces that exist as part of the custom legacy inventory management system.

Although this Web service has been valuable for various assessment purposes, once architects take a closer look at the actual Web service contract code, they detect some content that raises concerns:

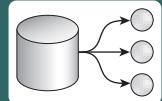
- The Web service operations inherited the cryptic legacy component method names.
- Several of the Web service operations have input and output message schemas that are derived from input and output legacy method parameters that are too granular for message-based service interaction.
- There is no real concept of an inventory record because it was not supported within the legacy component API.

These and other issues prompt Cutit to move ahead with a formal design process that requires the definition of service contracts prior to the development of underlying logic. This design process is completed subsequent to a formal analysis and modeling process during which architects collaborate with business analysts to define conceptual service candidates. These candidates then form the basis of the physical service designs.

Architects and developers can now avoid irregularities and problematic characteristics within service contracts because they have gained control of the definition of these contracts.

Metadata Centralization

How can service metadata be centrally published and governed?



Problem	Project teams, especially in larger enterprises, run the constant risk of building functionality that already exists or is already in development, resulting in wasted effort, service logic redundancy, and service inventory denormalization.
Solution	Service metadata can be centrally published in a service registry so as to provide a formal means of service registration and discovery.
Application	A private service registry needs to be positioned as a central part of an inventory architecture supported by formal processes for registration and discovery.
Impacts	The service registry product needs to be adequately mature and reliable, and its required use and maintenance needs to be incorporated into all service delivery and governance processes and methodologies.
Principles	Service Discoverability
Architecture	Enterprise, Inventory

Table 10.2

Profile summary for the Metadata Centralization pattern.

Problem

When growing a service inventory and fostering fundamental qualities such as those realized by Service Normalization (131) and Logic Centralization (136), there is a constant risk of project teams inadvertently (or sometimes even intentionally) delivering new services or service capabilities that already exist or are already in development (Figure 10.4).

This leads to undesirable results, most notably:

- the introduction of redundant service logic, which runs contrary to Logic Centralization (136)
- the introduction of overlapping service contexts, which runs contrary to Service Normalization (131)

- an overall less effective service inventory and technology architecture, bloated and convoluted by the added redundancy and denormalization and in need of additional governance effort

All of these characteristics can undermine an SOA initiative by reducing its strategic benefit potential.

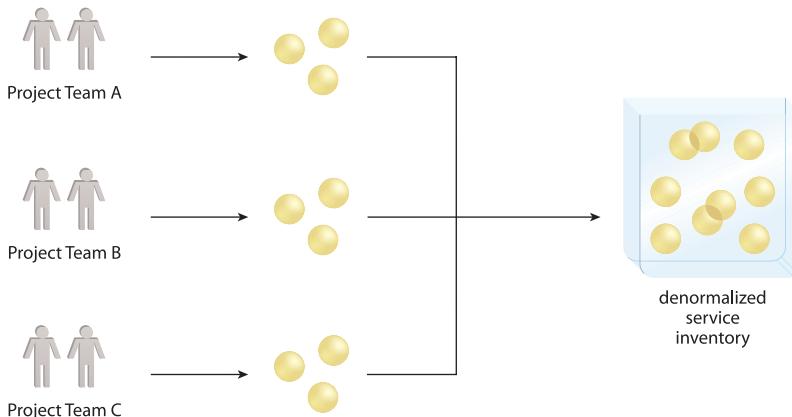


Figure 10.4

Without an awareness of the full range of existing and upcoming services, there is a constant risk that project teams will deliver service logic that already exists or is already in development.

Solution

A service registry is established as a central part of the surrounding infrastructure and is used by service owners and designers to:

- register existing services and capabilities
- register services and capabilities in development

As emphasized in discovery-related governance patterns, the registration process requires that discovery information be recorded in a highly descriptive and communicative manner so that it can be used by project teams to:

- locate and interpret existing services and learn about their functional contexts and boundaries
- locate and interpret service capabilities and learn about their invocation and interaction requirements

By providing a current and well-maintained registry of service contexts and capabilities, effective service discovery can be achieved (Figure 10.5).

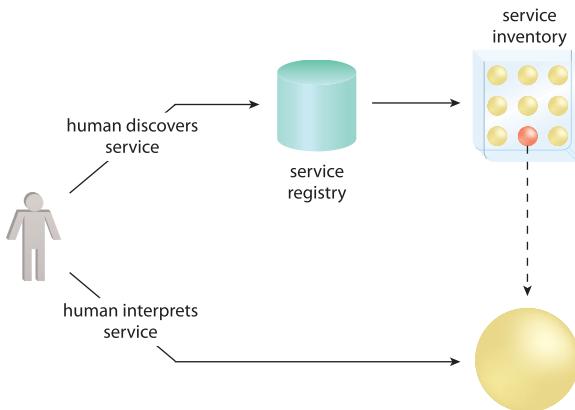


Figure 10.5

The fundamental discovery process during which a human locates a potential service via a service registry representing the service inventory and then interprets the service to determine its suitability.

NOTE

Metadata Centralization is clearly a design pattern associated with the Service Discoverability design principle and the discovery of services in general. Why then is it not simply called Service Discovery?

Service discovery itself is a process that is carried out once an enterprise has successfully applied Metadata Centralization to its architecture and the Service Discoverability design principle to its services. The process of service discovery is therefore related to a set of SOA governance patterns documented separately in the upcoming title *SOA Governance* that will be released as part of this book series.

Application

The application of this pattern requires the following common steps:

1. Regularly apply the Service Discoverability principle to all service contracts being modeled and designed.
2. Use service profiles and supporting processes to standardize the documentation of service and capability metadata. For example, a common part of service profiles is a standard vocabulary used for keywords that are attached to the service registry records.

3. Implement a reliable service registry product and position it as a standard part of the supporting infrastructure.

Finally, formal processes for the registration and discovery of services and capabilities need to be established.

NOTE

This pattern can be applied to a single service inventory or multiple domain inventories, depending on the ability of the service registry product to associate domains with service profile records. For a service profile template and descriptions of service discovery and interpretation processes, see Chapters 16 and 12, respectively, in *SOA Principles of Service Design*.

Impacts

Service registration and discovery processes are key success factors for the effective governance of a service inventory. If the processes are not respected or followed consistently by project teams or if the registry is not kept current, then the value potential of Metadata Centralization will severely diminish.

From a design perspective, however, this pattern will introduce the need for metadata standardization, as per the Service Discoverability principle. It will further require that metadata documentation and registration become part of the standard service delivery lifecycles.

There may further be a need to create a new organizational role in support of realizing Metadata Centralization. A person or a group would act as service registry custodian and assume responsibility for collecting the required metadata and maintaining the registry.

Relationships

Metadata Centralization essentially establishes a service registry, which is key to ensuring the long-term successful application of Logic Centralization (136) and Contract Centralization (409). If the correct services and their contracts can be effectively located (discovered), then the risk of inadvertently introducing redundant logic into an environment is reduced, further supporting Service Normalization (131).

Agnostic services represent the primary type of service for which metadata needs to be centralized for discovery purposes, which is why this pattern is especially relevant to services defined as a result of Entity Abstraction (175) and Utility Abstraction (168).

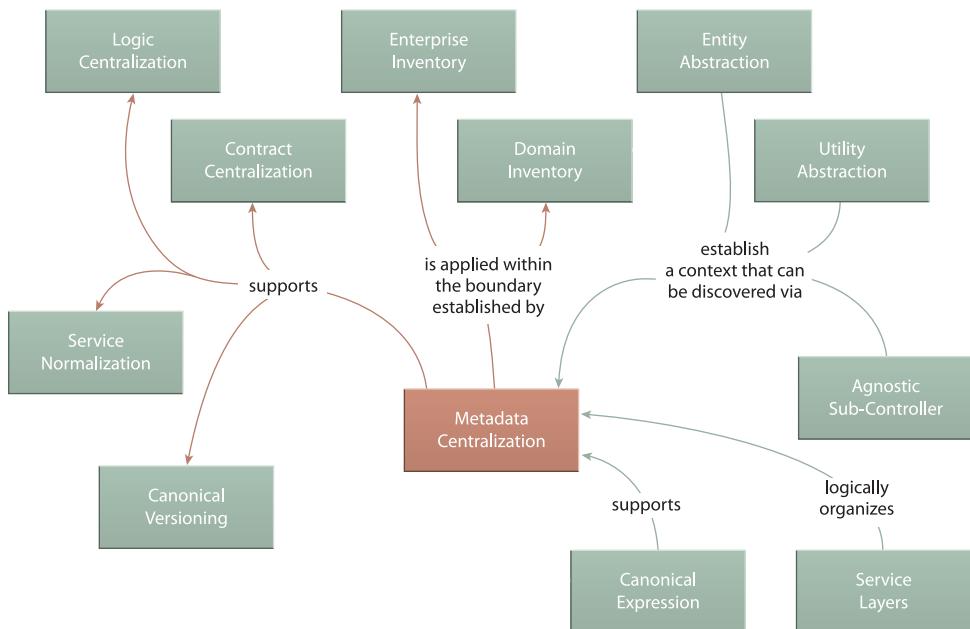


Figure 10.6

Metadata Centralization facilitates discovery and therefore relates to other patterns that rely on design-time awareness in order to be consistently applied.

CASE STUDY EXAMPLE

As explained in the Logic Centralization (136) example from Chapter 6, the original functional overlap between the Alleywood Areas and Region services could have gone undetected, resulting in the quality and integrity of the service inventory being negatively affected. For this reason, it was determined early on that a service registry would be required to support Service Normalization (131) and ensure the consistent application of Logic Centralization (136).

However, due to the decision to establish separate domain service inventories, architects struggle with the option of implementing a separate service registry for each inventory. Although it would continue to allow each group to govern their respective service collection independently, it would establish two different repositories.

It is anticipated that Alleywood and Tri-Fold services will need to interoperate. Those creating cross-inventory compositions will therefore need to issue separate queries in order to discover the required service capabilities.

The awkwardness of this governance architecture eventually prompts McPherson to establish a central enterprise service registry instead (Figure 10.7). This registry is governed by the McPherson Enterprise Group and allows Alleywood and Tri-Fold project teams to search each others' inventories.

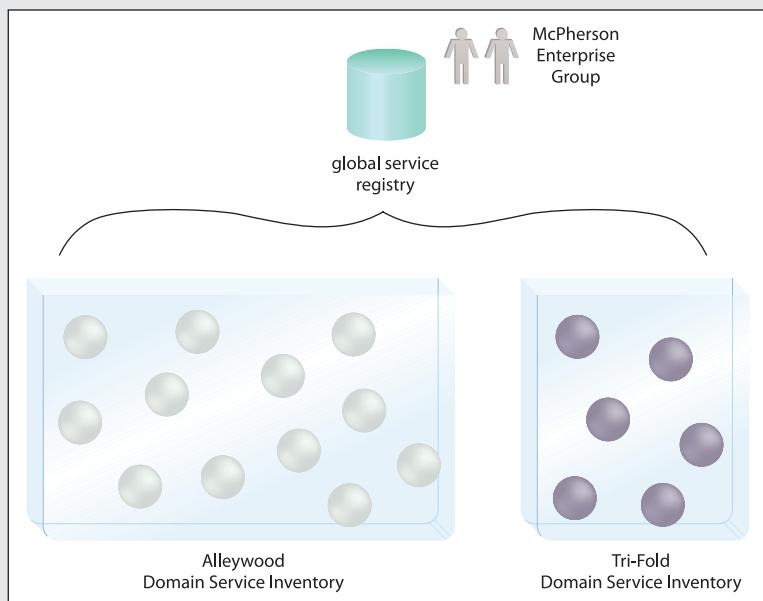
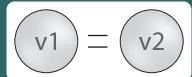


Figure 10.7

A global registry that spans services across Alleywood and Tri-Fold inventories.

Canonical Versioning

How can service contracts within the same service inventory be versioned with minimal impact?



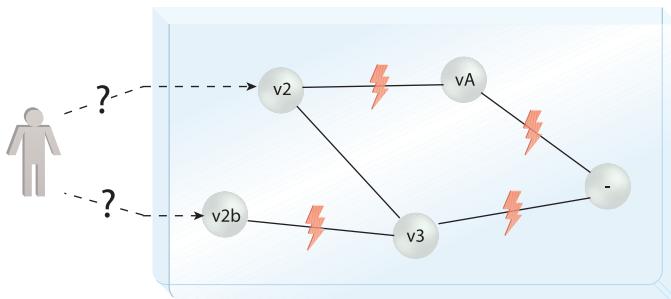
Problem	Service contracts within the same service inventory that are versioned differently will cause numerous interoperability and governance problems.
Solution	Service contract versioning rules and the expression of version information are standardized within a service inventory boundary.
Application	Governance and design standards are required to ensure consistent versioning of service contracts within the inventory boundary.
Impacts	The creation and enforcement of the required versioning standards introduce new governance demands.
Principles	Standardized Service Contract
Architecture	Service, Inventory

Table 10.3

Profile summary for the Canonical Versioning pattern.

Problem

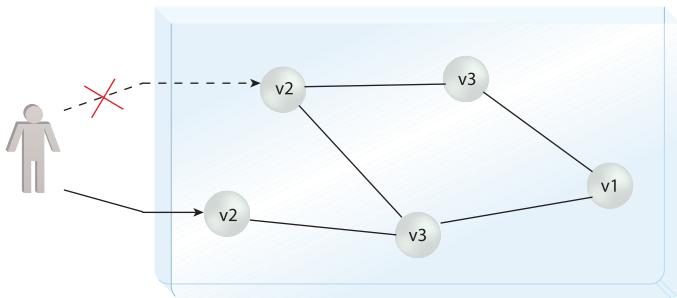
When service contracts within the same service inventory are subjected to different versioning approaches and conventions, post-implementation contract-level disparity emerges, compromising interoperability and effective service governance (Figure 10.8). This can negatively impact design-time consumer development, runtime service access, service reusability, and the overall evolution of the service inventory as a whole.

**Figure 10.8**

Services that have been versioned differently become challenging to compose and interoperate and also difficult to interpret.

Solution

Service contracts within the same inventory are versioned according to the same conventions and as part of the same overall versioning strategy (Figure 10.9). This ensures a consistent governance path for each service, thereby preserving contract standardization and intra-inventory compatibility and interoperability.

**Figure 10.9**

When services are versioned according to the same overarching strategy, they can retain their original standardization and interoperability and are more easily understood by consumer designers.

Application

This pattern generally requires that a single versioning strategy be chosen, comprised of a series of rules and conventions that essentially become governance standards.

Canonical Versioning approaches can vary depending on the complexion of the enterprise, existing versioning or configuration management methodologies that may already be in

place, and the nature of the overall governance strategy that may have also been established.

There are three common strategies that provide a baseline set of rules:

- *Strict* – Any compatible or incompatible change results in a new version of the service contract. This approach does not support backwards or forwards compatibility and is most commonly used when service contracts are shared between partner organizations and when changes to a contract can have legal implications.
- *Flexible* – Any incompatible change results in a new version of the service contract, and the contract is designed to support backwards compatibility but not forwards compatibility.
- *Loose* – Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility and forwards compatibility.

NOTE

The terms “backwards compatibility” and “forwards compatibility” are explained in the description for Compatible Change (465) in Chapter 16. For examples of each of these versioning strategies, see Chapters 20–23 in *Web Service Contract Design and Versioning for SOA*.

Impacts

There is the constant risk that project teams will continue to use their own versioning approaches, or rely too heavily on patterns like Concurrent Contracts (421), which allows them to simply add new contracts to an existing service.

The successful application of any versioning strategy will require strong support for the adherence to its rules and conventions to the extent that the chosen versioning approach becomes an inventory-wide standard on par with any other design standard. This introduces the need for a new organizational role that is tasked with enforcing the processes and syntactical characteristics that are defined as part of the strategy.

Relationships

Canonical Versioning essentially formalizes the application of Compatible Change (465), Version Identification (472), and Termination Notification (478), in that the overarching

strategy established by this pattern will determine how and to what extent each of these more specific versioning patterns is applied.

The application of Metadata Centralization (280) results in a service registry that enables effective discovery of different contract versions and Canonical Expression (275) implements characteristics in service contracts that improve their legibility. Both of these patterns therefore aid the goals of Canonical Versioning.

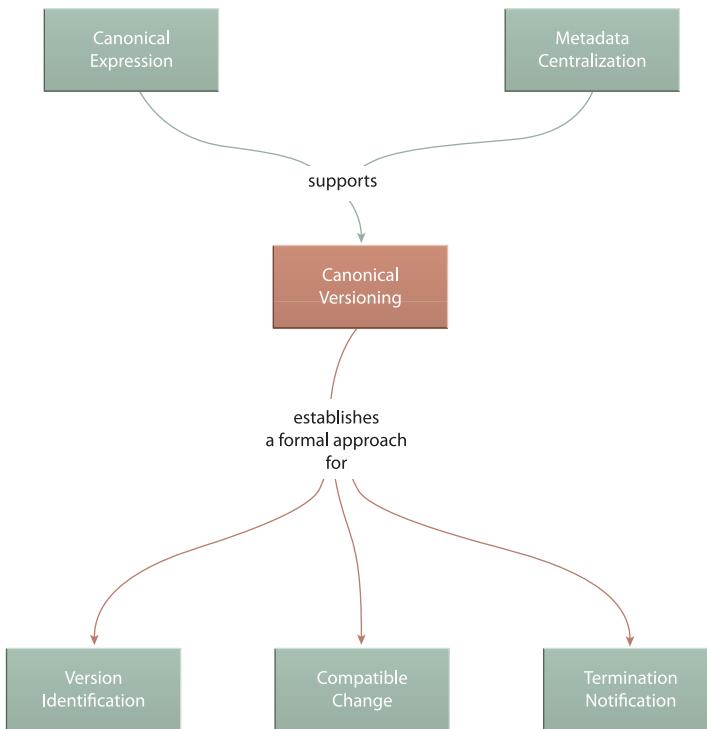


Figure 10.10

Canonical Versioning is primarily related to other versioning patterns.

NOTE

Before continuing with this case study example, be sure to read up on the Policy Check service that was defined in the State Repository (242) case study example and then later positioned to support multiple inventories as part of the example for Cross-Domain Utility Layer (267).

CASE STUDY EXAMPLE

The FRC announced that due to new government legislation, it has revised some of its policies. This changes the policy data that was being made available electronically via its public Web services.

The Alleywood Policy Check service was originally positioned to shield the rest of the Alleywood service inventory from these types of changes by providing the sole access point for FRC policy data. Although its service logic can be augmented to accommodate changes to the FRC services, architects soon realize that they cannot prevent having to issue a new version of the Policy Check contract because the FRC has added new content and structure into their policy schemas.

Being the first time they've had to contend with a major versioning issue, the Alleywood team decides that some formal approach needs to be in place before they proceed. After some research into common versioning practices and further deliberation, they produce a versioning strategy comprised of a set of specific conventions and rules:

Version Identification (472) will be applied as follows:

- Version information will be expressed in major numbers displayed left of the decimal point and minor version numbers displayed to the right of the decimal point (e.g., “1.0”).
- Minor and major contract version numbers will be expressed using the WSDL documentation element by displaying the word “Version” before the version number (e.g., <documentation>Version 1.0</documentation>)
- Major version numbers will be appended to the WSDL definition’s target namespace and prefixed with a “v” as shown here: <http://alleywoodlumber/contract/po/v1>

Compatible Change (465) will be applied as follows:

- A compatible change in the WSDL definition increments the minor version number and does not change the WSDL definition target namespace.
- A compatible change in the XML Schema definition increments the minor version number and does not change the XML Schema or WSDL definition target namespaces.
- An incompatible change in the WSDL definition increments the major version number and forces a new WSDL target namespace value.

- An incompatible change in the XML Schema definition increments the major version number and forces a new target namespace value for both the XML Schema and WSDL definitions.

The previously described scenario results in a set of incompatible changes that requires that the major version number of the Policy Check service contract be incremented from 1.0 to 2.0.

This example demonstrates the beginning of the Policy Check XML Schema and WSDL definitions after this change has occurred:

```
<xsd:schema xmlns:xsd=
  "http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://alleywoodlumber/schema/pc/v2"
  xmlns="http://alleywoodlumber/schema/pc/v2"
  version="2.0">
  <xsd:annotation>
    <xsd:documentation>
      Version 2.0
    </xsd:documentation>
  </xsd:annotation>
  ...
</xsd:schema>

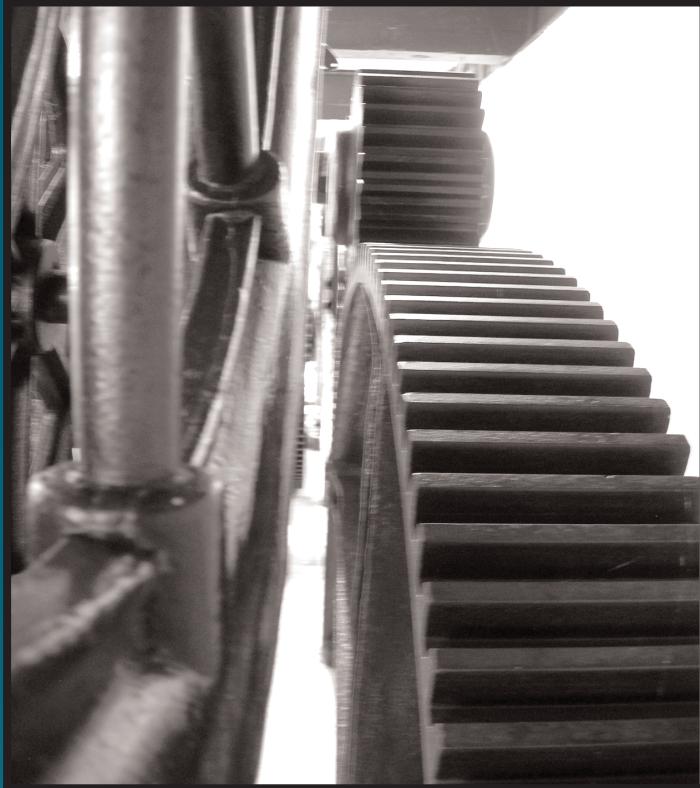
<definitions name="Policy Check" targetNamespace=
  "http://alleywoodlumber/contract/pc/v2"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://alleywoodlumber/contract/pc/v2"
  xmlns:pc="http://alleywoodlumber/schema/pc/v2"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <documentation>
    Version 2.0
  </documentation>
  ...
</definitions>
```

Example 10.1

Fragments from the Policy Check Web service contract documents that show the effects of applying a versioning strategy.

The Alleywood architects acknowledge that defining the versioning approach is only the first step. In order for Canonical Versioning to be fully realized, these new rules and standards must be applied to any future service contracts that need to be versioned. This leads to the creation of a new process that is placed under the jurisdiction of the governance group.

This page intentionally left blank



Part III

Service Design Patterns

Chapter 11: Foundational Service Patterns

Chapter 12: Service Implementation Patterns

Chapter 13: Service Security Patterns

Chapter 14: Service Contract Design Patterns

Chapter 15: Legacy Encapsulation Patterns

Chapter 16: Service Governance Patterns

This page intentionally left blank

Chapter 11



Foundational Service Patterns

Functional Decomposition

Service Encapsulation

Agnostic Context

Non-Agnostic Context

Agnostic Capability

The design patterns in this chapter represent the most essential steps required to partition and organize solution logic into services and capabilities in support of subsequent composition. In many ways, these patterns can be considered fundamental service-orientation theory.

As shown in Figure 11.1, the patterns are organized into a proposed application sequence. As much as the individual patterns provide proven solutions, the suggested application sequence itself is also proven, which is why this chapter is structured accordingly.

The patterns are divided into two groups that lead up to the application of the composition patterns in Chapter 17, as follows:

1. *Service Identification Patterns* – The overall solution logic required to solve a given problem is first defined, and the parts of this logic suitable for service encapsulation are subsequently filtered out.
2. *Service Definition Patterns* – Base functional service contexts are defined and used to organize available service logic. Within agnostic contexts, service logic is further partitioned into individual capabilities.
3. *Capability Composition Patterns* – The previous patterns establish the boundaries of capability utilization, which naturally leads to the composition patterns described in Chapter 17.

When you string the service identification and service definition patterns together in their suggested application sequence, you will end up with a sequence of steps that can be considered a primitive service modeling process. The purpose of this process is only to raise the more fundamental considerations when shaping services into candidates for subsequent design. In real-life applications, the steps represented by these patterns would be part of a larger, customized process and methodology.

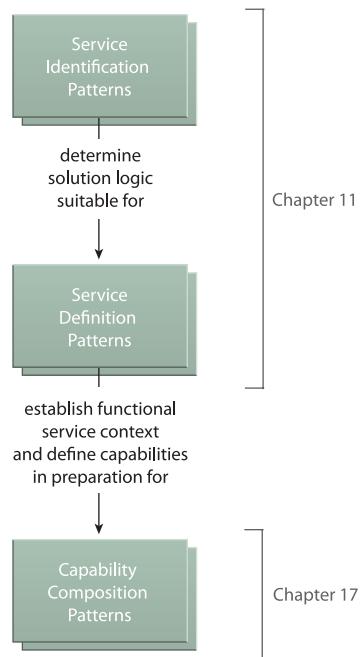


Figure 11.1

The patterns in this chapter follow a sequence that is carried over in Chapter 17.

NOTE

If you haven't already, be sure to also read the *How Foundational Inventory and Service Design Patterns Relate* section at the beginning of Chapter 6.

CASE STUDY BACKGROUND

Due to the emphasis on the fundamental process established by the pattern sequence, all case study examples in this chapter revolve around a simple business task presenting a modest set of requirements that are fulfilled through the application of these patterns.

The following process description provides the necessary background information for subsequent examples.

The Chain Inventory Transfer Business Process

Whenever Cutit Saws manufactures a new chain, it undergoes a short process that takes the newly assembled chain from the manufacturing to the inventory control departments, as follows:

1. The assembled chain is released by the manufacturing team and delivered to a Quality Inspector.
2. The chain undergoes a manual inspection for defects, which includes endurance and alignment tests.
3. If the chain passes these tests, it is forwarded to the Inventory Controller (a person). If any one of the tests fail, the chain is sent back to the manufacturing team, along with a test report that is not electronically recorded.
4. The Inventory Controller generates an inventory record for the chain to officially add it to the inventory.
5. As part of creating an inventory record, the chain is assigned a predefined model number. If back orders for this model exist, the chain is also associated with the next back order in line.
6. As part of a back order, the chain is then forwarded to the shipping department along with a corresponding order document.
7. If the chain model is not on back order, the chain is simply added to the existing inventory stock.

As illustrated in Figure 11.2, this process consists of a combination of automated and manual steps.

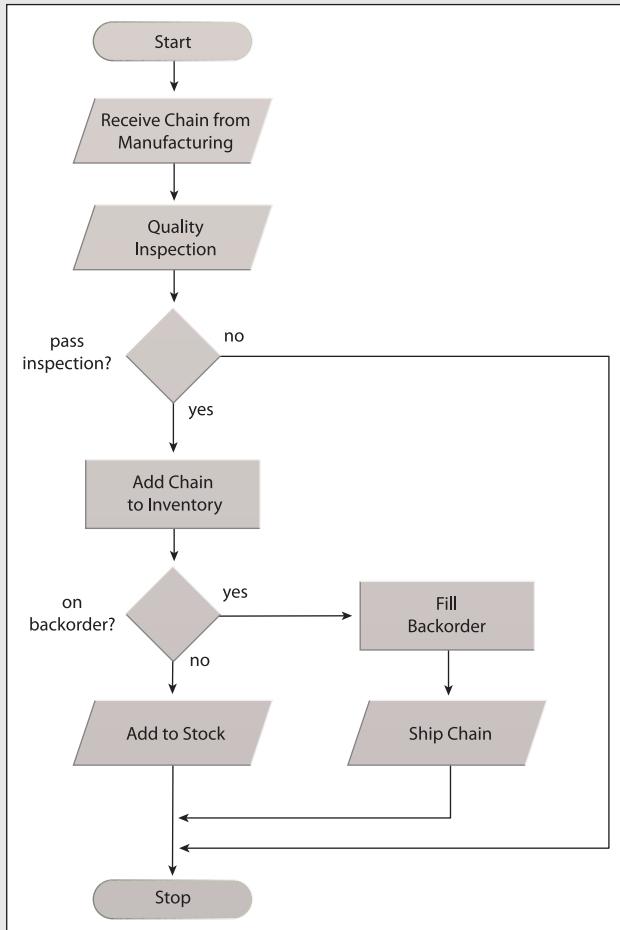


Figure 11.2

The Chain Inventory Transfer business process transitions a newly manufactured chain to the inventory warehouse from where it is either shipped or stored.

NOTE

This case study example continues into Chapter 17.

11.1 Service Identification Patterns

These initial design patterns (Figure 11.3) essentially carry out a separation of concerns in support of service-orientation during which solution logic is decomposed and the portions suitable for service encapsulation are identified. The result is a foundation of unorganized logic ready to be shaped into legitimate services via the application of the subsequent service definition patterns and the principles of service-orientation.

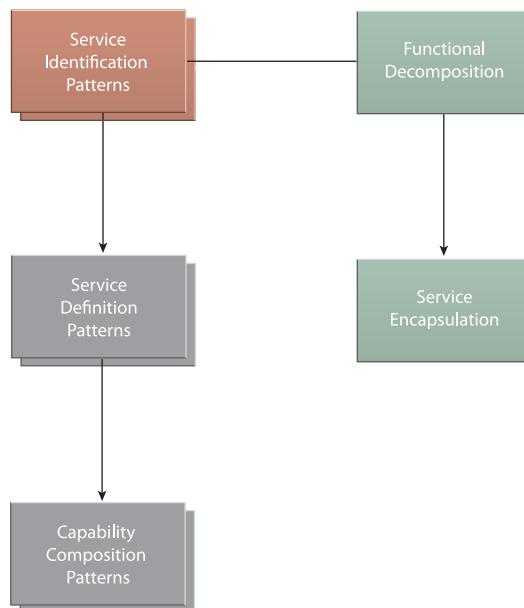


Figure 11.3

The two patterns in this section raise specific considerations that apply to the process of carrying out a separation of concerns.

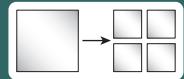
NOTE

You might notice that these upcoming patterns are the only two design patterns in this book that are not directly related to any service-orientation design principles (as per the *Principles* field in the upcoming profile summary tables).

These patterns are so foundational that there is not yet an opportunity to involve or connect them with the specific design considerations raised by common service-orientation principles. However, one could argue that because they are so fundamental to establishing services that they are, in fact, related to all parts of service-orientation.

Functional Decomposition

How can a large business problem be solved without having to build a standalone body of solution logic?



Problem	To solve a large, complex business problem a corresponding amount of solution logic needs to be created, resulting in a self-contained application with traditional governance and reusability constraints.
Solution	The large business problem can be broken down into a set of smaller, related problems, allowing the required solution logic to also be decomposed into a corresponding set of smaller, related solution logic units.
Application	Depending on the nature of the large problem, a service-oriented analysis process can be created to cleanly deconstruct it into smaller problems.
Impacts	The ownership of multiple smaller programs can result in increased design complexity and governance challenges.
Principles	n/a
Architecture	Service

Table 11.1

Profile summary for the Functional Decomposition pattern.

Problem

Most business tasks or business processes requiring automation constitute large problems. An accepted approach to solving a large automation problem has been to build an application. Prior to the advent of distributed computing, custom-developed applications were primarily designed as monolithic executables—single, self-contained bodies of solution logic (Figure 11.4).

Repeatedly solving large problems by building monolithic solution logic results in an enterprise comprised of single-purpose applications residing in siloed implementation boundaries.

For many organizations such environments have posed significant challenges associated with extensibility and cross-application connectivity. Furthermore, a siloed technology landscape can become bloated and expensive to maintain and change—so much so that

many of these applications have remained in modernized technical environments as entrenched legacy systems that continue to inhibit the overall evolution of the enterprise.

Solution

Functional Decomposition is essentially an application of the separation of concerns theory. This established software engineering principle promotes the decomposition of a larger problem into smaller problems (called *concerns*) for which corresponding units of solution logic can be built.

The rationale is that a larger problem can be more easily and effectively solved when separated into smaller parts. Each unit of solution logic that is built exists as a separate body of logic responsible for solving one or more of the identified, smaller concerns (Figure 11.5). This design approach is well-established and forms the basis for previous and current distributed computing platforms.

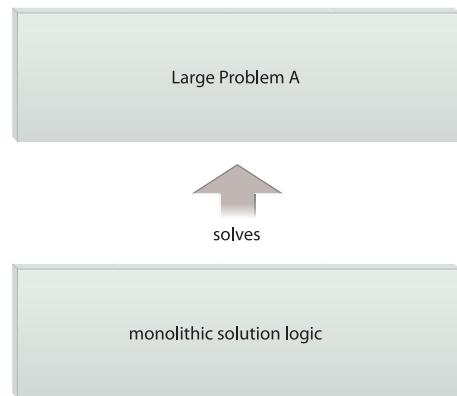


Figure 11.4

One approach to solving a large problem is to build a correspondingly large body of solution logic.

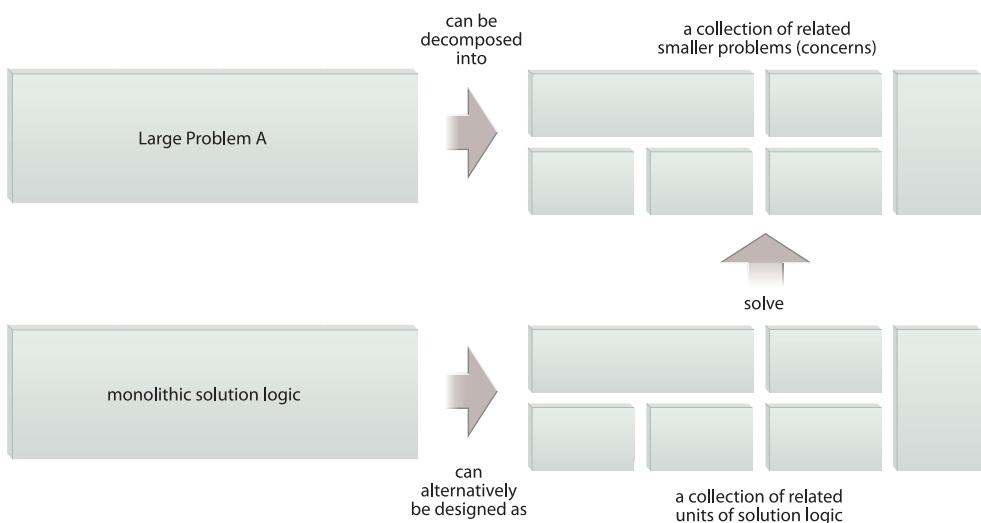


Figure 11.5

Distributed computing is based on an approach where a large problem is decomposed and its corresponding solution logic is distributed across individual solution logic units. On its own, this pattern results in the decomposition of the larger problem into smaller problems, as shown in the top part of this figure. The actual definition of solution logic units occurs through the application of subsequent patterns in this chapter.

NOTE

One of the key considerations when applying this pattern as part of the overall application sequence is that many of the individually defined units of solution logic will eventually be expected to be able to help solve additional large problems in order to achieve the target state explained in the pattern description for Capability Recomposition (526) and illustrated in Figure 17.5.

Application

As previously stated, Function Decomposition is essentially realized by carrying out the separation of concerns in support of service-orientation. A primary means by which service-orientation is distinguished from other distributed design approaches is the manner in which separation is achieved and how units of solution logic are defined.

This pattern is therefore not applied independently. It represents the starting point for a process that begins with functional separation and then continues through to shape separated logic into services, as per the subsequent patterns in this chapter.

In practice, this form of decomposition is generally achieved via a service modeling process that begins with a preliminary identification of individual concerns. The large problem corresponds to a business process that needs to be automated. The functional decomposition of this business process results in the definition of granular process steps, each of which can be considered an individual concern.

Impacts

Distributed units of solution logic require individual attention with regards to interconnectivity, security, reliability, and maintenance in order to ensure that each chain in the link of runtime activity processing is and remains adequately reliable and self-sufficient. An environment consisting of a large amount of smaller software programs therefore imposes more design complexity and governance challenges than one comprised of a single monolithic application.

Furthermore, the effectiveness of this pattern is limited by the quality of the problem definition. For a business process (representing the larger problem) to be properly decomposed, it needs to be documented in an accurate and detailed manner so that individual process steps are sufficiently granular. If the quality of the business process definition is poor, then the resulting concerns will form a weak foundation for subsequent service definition.

Relationships

On a fundamental level, you could say that Functional Decomposition forms the basis for all of the patterns in this book. But when identifying direct relationships, the only pattern that really qualifies is Service Encapsulation (305). Functional Decomposition essentially prepares the concerns that are subsequently addressed by solution logic that begins to take shape with the application of Service Encapsulation (305).

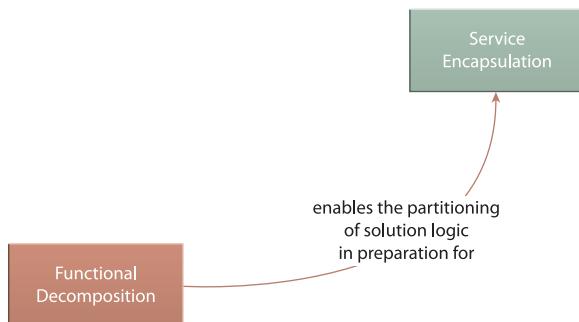


Figure 11.6

This displayed relationship simply establishes how the reasoning behind decomposing functionality is to make the decomposed parts available for potential encapsulation by services.

CASE STUDY EXAMPLE

The requirement for the Chain Inventory Transfer process to be established and automated wherever appropriate represents a large business problem. Using the process description documented previously in Figure 11.2 as a starting point, the Cutit team decomposes this problem into a set of concerns roughly equivalent to the primary process steps, as follows:

Concern 1 – How can the chain's quality be guaranteed?

Concern 2 – How can the chain be recorded as part of the inventory?

Concern 3 – How can the chain be cross-referenced with any corresponding back orders?

Concern 4 – How can the chain be associated with the correct back order?

Concern 5 – How can the chain be delivered to fulfill the back order?

(Note that in larger business process definitions it is more common to bundle groups of related, more granular steps to represent individual concerns.)

For each of these concerns, corresponding solutions are defined:

Solution for Concern 1 – The chain is manually inspected.

Solution for Concern 2 – The chain is electronically recorded as part of the inventory control system.

Solution for Concern 3 – Upon recording the chain, the system performs a cross-check for corresponding back orders, using the model number as the search criteria.

Solution for Concern 4 – The next back order in the queue is pulled up from the order management system and filled using the chain's inventory record.

Solution for Concern 5 – The chain is manually shipped.

In many cases, the identification of concerns can occur prior to the definition of the actual business process workflow. In this case, the concerns are represented by a set of related business requirements that are solved via the execution of steps within a larger business process definition.

Service Encapsulation

How can solution logic be made available as a resource of the enterprise?



Problem	Solution logic designed for a single application environment is typically limited in its potential to interoperate with or be leveraged by other parts of an enterprise.
Solution	Solution logic can be encapsulated by a service so that it is positioned as an enterprise resource capable of functioning beyond the boundary for which it is initially delivered.
Application	Solution logic suitable for service encapsulation needs to be identified.
Impacts	Service-encapsulated solution logic is subject to additional design and governance considerations.
Principles	n/a
Architecture	Service

Table 11.2

Profile summary for the Service Encapsulation pattern.

Problem

A collection of related software programs that represent a larger, decomposed body of solution logic can continue to exist within a siloed application boundary. In fact, many past distributed systems were built this way. The decision to partition the solution logic into smaller units was often motivated by the following considerations:

- increasing scalability by separating the parts of the system more subject to high volume and concurrency
- improving security by isolating specific parts of the system with special access and privacy requirements
- increasing reliability by distributing critical parts of a system across multiple physical servers
- achieving nominal reuse within the system boundary (or within a limited part of the enterprise)

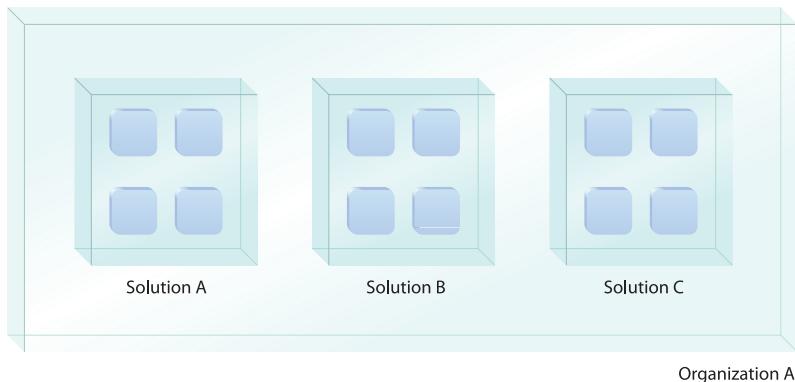


Figure 11.7

An enterprise consisting of distributed, yet still siloed solutions.

When an enterprise is comprised of siloed (or quasi-siloed) distributed solutions (Figure 11.7), it can encounter many design and governance challenges, such as:

- significant amounts of waste and redundancy
- inefficient application delivery
- bloated, oversized technical environments
- complex infrastructure and convoluted enterprise architecture
- complex and expensive integration
- ever-increasing IT operational costs

Details regarding these issues are documented in the *Life Before Service-Orientation* section of *SOA Principles of Service Design* and also at SOAPrinciples.com.

Solution

Solution logic suitable for classification as an enterprise resource can be encapsulated by and exposed as a service. This essentially means that the logic itself may form the basis for a new service, or the logic may be encapsulated by an existing service (most likely as a new capability). This results in an environment where services are shared (Figure 11.8).

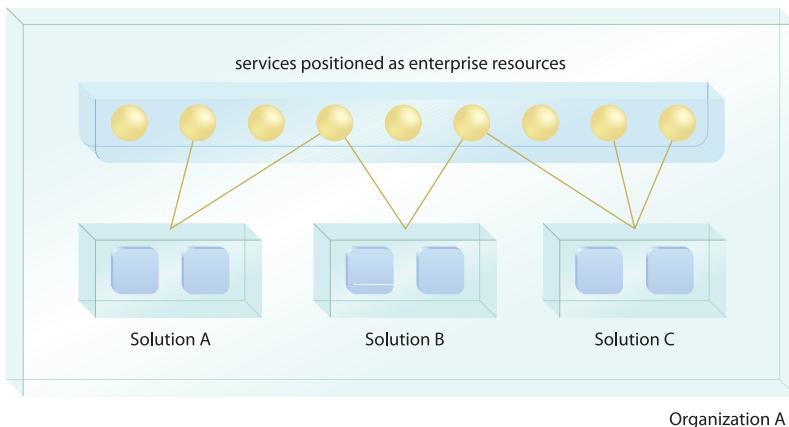


Figure 11.8

An enterprise wherein individual solutions use logic encapsulated as services and vice versa.

Application

The first required step is to identify and filter out solution logic that is actually suitable as an enterprise resource. Not all solution logic falls into this category. There will be bodies of logic that are tailored for individual distributed applications and for which other design approaches may be more appropriate.

Here are some guidelines:

- *Does the logic contain functionality that is useful to parts of the enterprise outside of the immediate application boundary?*

If it does, the logic has increased value potential that may warrant its classification as an enterprise resource. This type of logic generally forms the basis of an agnostic service, as per Agnostic Context (312).

- *Does logic designed to leverage enterprise resources also have the potential to become an enterprise resource?*

This form of logic emerges after evident agnostic logic is initially separated. It may be required for service-orientation to be applied to this type of logic so that it remains uniform with agnostic services and so that some or all of its functionality can also be positioned as an enterprise resource. This option is further explored in Non-Agnostic Context (319).

- Does the implementation of the logic impose hard constraints that make it impractical or impossible to position the logic as an effective enterprise resource?

Regardless of whether the nature of the logic makes it suitable as an enterprise resource, there may be real-world limitations that prevent it from being effectively encapsulated by a service.

Using criteria such as this, the solution logic suitable for service encapsulation can be identified, allowing unsuitable logic to be filtered out (Figure 11.9).

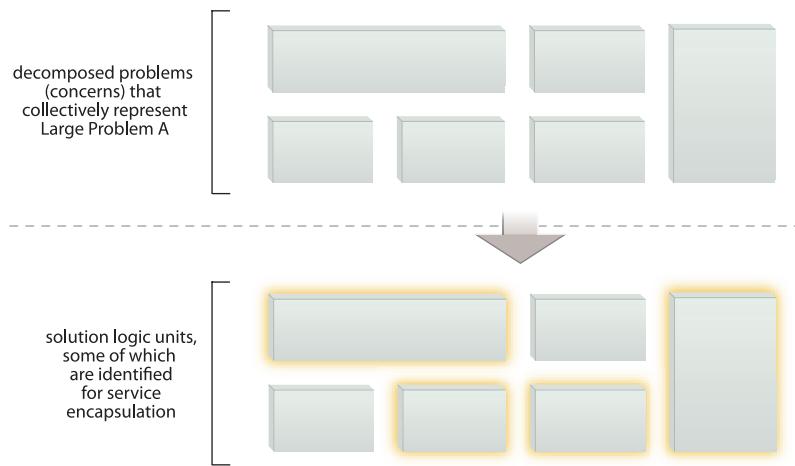


Figure 11.9

A subset of the decomposed monolithic solution logic is identified as being suitable for service encapsulation (as represented by the highlighted blocks).

For encapsulated solution logic to become an effective member of a service inventory, it needs to be further shaped by other patterns and principles so that it is designed to support the strategic goals associated with service-oriented computing.

A solid knowledge of the service-orientation design paradigm is therefore necessary in order to best determine when logic is and is not suitable for service encapsulation. As a rule of thumb, if service-orientation design principles cannot be applied to a meaningful extent, the logic will not likely warrant service encapsulation.

As previously stated, how this logic is determined is based on the methodology used and the maturity of the existing service inventory. Logic identified as being suitable for service encapsulation may be assigned to an existing service, or it may form the basis of a new service.

Impacts

Because the application of this pattern results in the identification and filtering of logic (in preparation for the upcoming group of service definition patterns), there is no immediate impact.

However, it should be noted that its application is limited to the filtering process only. Logic that is not considered suitable for service encapsulation is given no further consideration by this or any other patterns in this chapter. Therefore, this pattern sequence must be part of a larger analysis process that encompasses the modeling of solution logic that will not be encapsulated within services.

Relationships

Logic deemed suitable for service encapsulation is subsequently grouped into single or multi-purpose services, as per Non-Agnostic Context (319) and Agnostic Context (312).

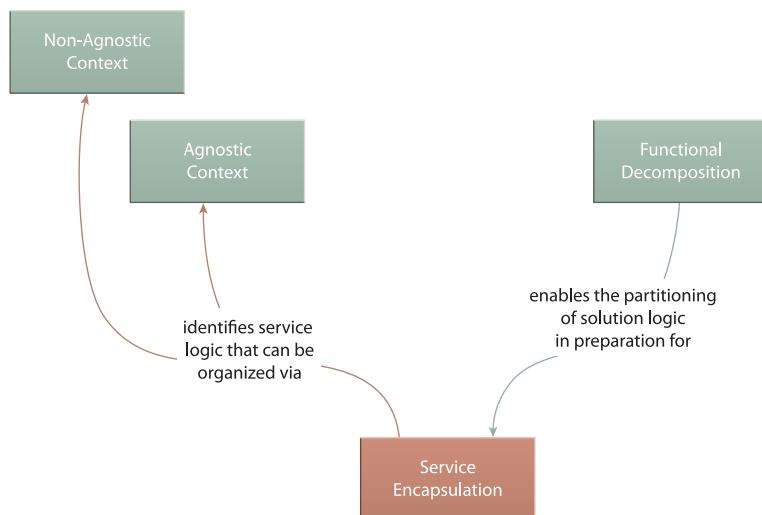


Figure 11.10

Service Encapsulation determines what logic will eventually comprise services.

CASE STUDY EXAMPLE

Upon reviewing the individual solutions defined as a result of applying Functional Decomposition (300), the Cutit team determines which parts are suitable for service encapsulation, as follows:

- Solutions for Concerns 1 and 5 are not suitable because they represent steps that are manually performed.
- Solutions for Concerns 2, 3, and 4 are suitable because they represent logic that can be automated and has no known limitations for it to be potentially useful on an enterprise basis.

Solutions for Concerns 1 and 5 remain part of the overall solution design—only they won't be delivered as services. The remaining solutions (for Concerns 2, 3, and 4) are further subject to the upcoming service definition patterns.

11.2 Service Definition Patterns

The identification of logic suitable for service encapsulation and the grouping and distribution of that logic within distinct functional contexts establishes fundamental service boundaries. These boundaries become increasingly important as an inventory of services is assembled and inventory-related patterns, such as Service Normalization (131), are applied to avoid functional overlap.

To define the most suitable boundary for a service requires that the most suitable functional context be established. This determines what functionality belongs within and outside of a service boundary. This next set of patterns (Figure 11.11) help make this determination by providing criteria for whether service logic is to be considered agnostic or non-agnostic and further guidance for how agnostic service logic in particular can be organized into separate capabilities.

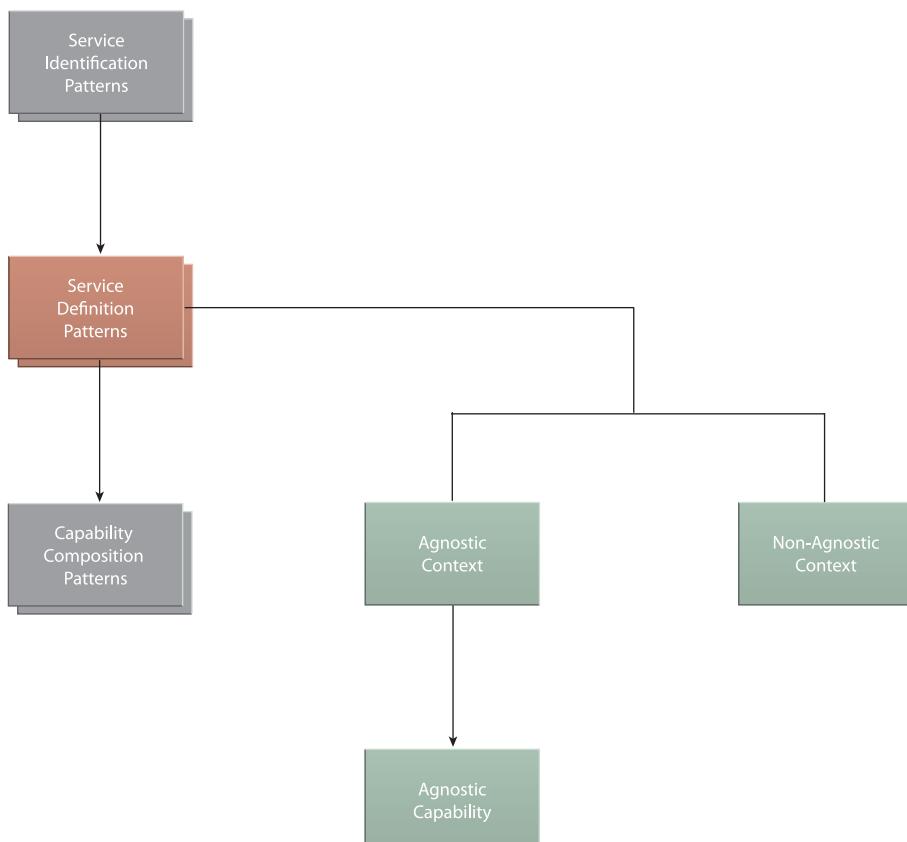
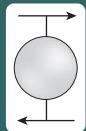


Figure 11.11

Service definition patterns organize service logic into specific contexts, thereby establishing service boundaries.

Agnostic Context

How can multi-purpose service logic be positioned as an effective enterprise resource?



Problem	Multi-purpose logic grouped together with single purpose logic results in programs with little or no reuse potential that introduce waste and redundancy into an enterprise.
Solution	Isolate logic that is not specific to one purpose into separate services with distinct agnostic contexts.
Application	Agnostic service contexts are defined by carrying out service-oriented analysis and service modeling processes.
Impacts	This pattern positions reusable solution logic at an enterprise level, potentially bringing with it increased design complexity and enterprise governance issues.
Principles	Service Reusability
Architecture	Service

Table 11.3

Profile summary for the Agnostic Context pattern.

NOTE

For a description of the term “agnostic” and related background information, see the *Agnostic Logic and Non-Agnostic Logic* section at the beginning of Chapter 7.

Problem

The solution logic required to solve a single concern will frequently include logic that is also suitable for solving other concerns. Grouping single and multi-purpose functionality together into one unit of logic will limit or even eliminate the potential for reuse (Figure 11.12).

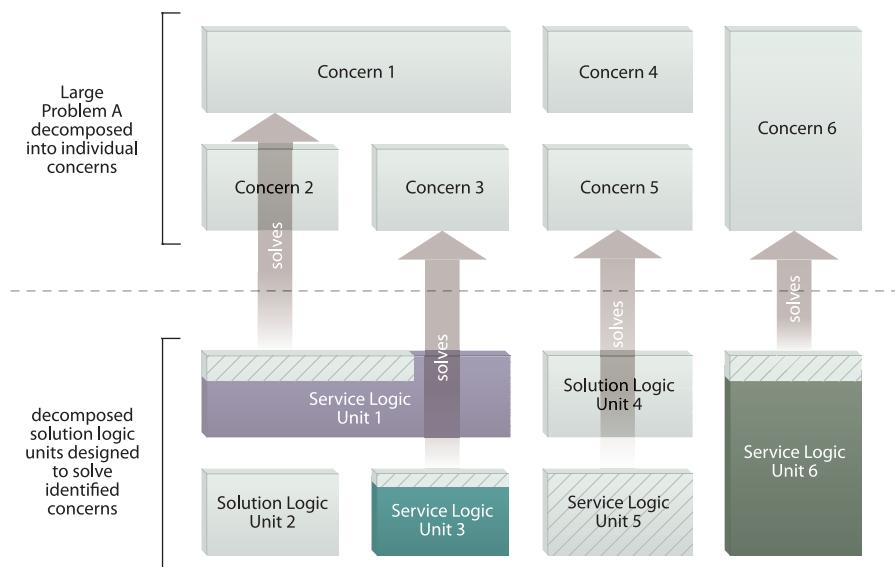


Figure 11.12

Decomposed units of solution logic will naturally be designed to solve concerns specific to a single, larger problem. Units 1, 3, and 6 represent logic that contains multi-purpose functionality trapped within a single-purpose (single concern) context. Single-purpose (non-agnostic) logic is represented by the striped pattern in this diagram.

Solution

Solution logic that is agnostic to the larger problem is separated from logic that is specific to the larger problem. One or more services with distinct agnostic functional contexts are then identified within which the agnostic logic is located (Figure 11.13).

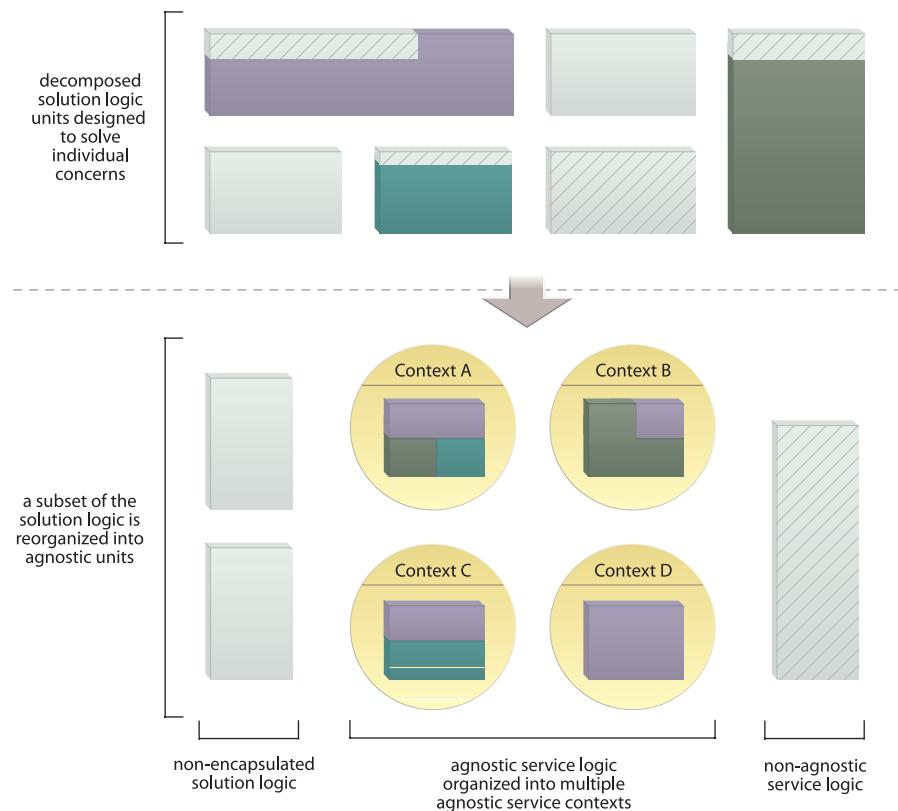


Figure 11.13

The application of this pattern results in a subset of the solution logic being further decomposed and then distributed into services with specific agnostic contexts.

Application

Solution logic is further decomposed and reorganized as a result of carrying out formal analysis and modeling processes. Agnostic logic is defined and continually refined into a set of candidate service contexts. These contexts can be based on pre-defined agnostic service model classifications, such as those that form the basis of Entity Abstraction (175) and Utility Abstraction (168).

Impacts

The application of this design pattern essentially results in the creation of services with reuse potential, which ties directly into several strategic service-oriented computing benefits, including an increased and repeatable return on investment.

Achieving these benefits tends to increase the overall quantity of services required to solve a given problem, which leads to additional design considerations and performance overhead associated with service compositions.

The governance effort of agnostic services is significantly more than if the corresponding solution logic was dedicated to a single application. Additionally, the governance of the overall architecture is also impacted as the quantity of agnostic services within an inventory grows.

Relationships

From a service design perspective, Agnostic Context is one of the most distinctive patterns associated with service-orientation. It therefore has several relationships with other patterns that apply specialized variations of Agnostic Context, such as Entity Abstraction (175) and Utility Abstraction (168). The closest relationship is between Agnostic Context and Agnostic Capability (324), as the latter is applied to services that have already been deemed agnostic.

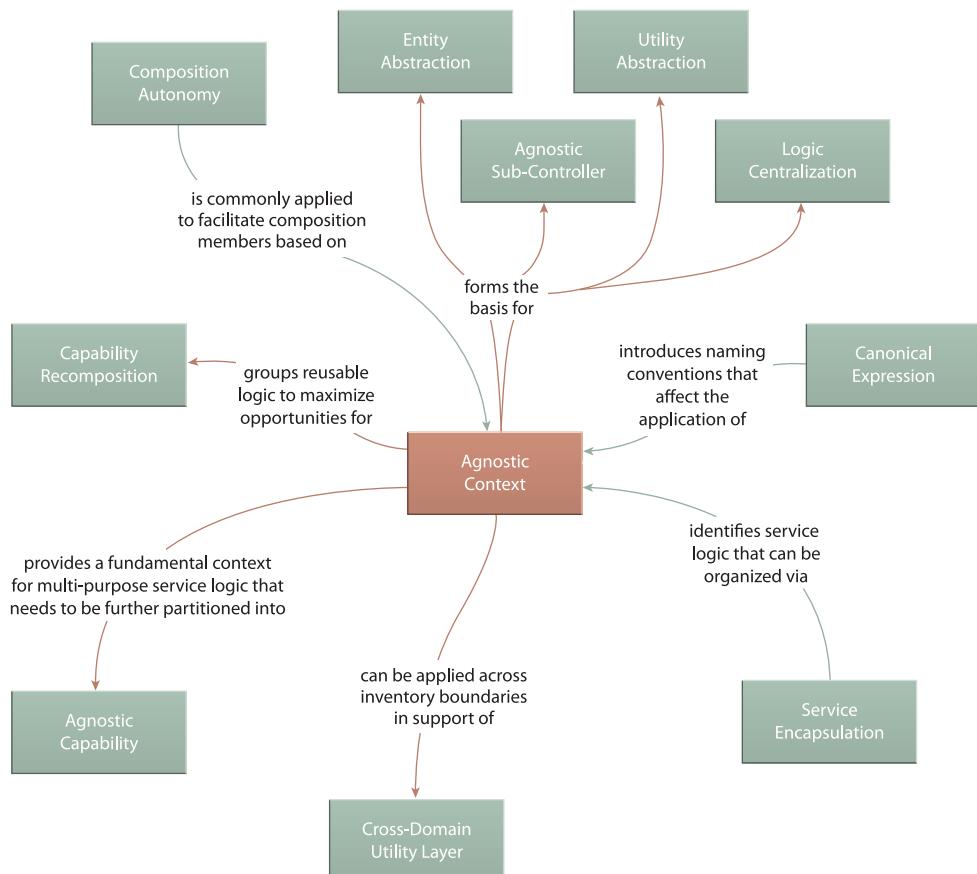


Figure 11.14

Agnostic Context is core to service design and is responsible for forming the basis of several fundamental architectural design patterns.

CASE STUDY EXAMPLE

Cutit analysts and architects review the logic identified as being potentially suitable for service encapsulation and break down this logic into individual actions, as follows:

Solution for Concern 2

- create a new inventory record
- retrieve chain specification information by serial number
- enter chain information, including the assignment of pre-defined model number
- save new inventory record

Solution for Concern 3

- issue a query for back orders keyed on model number
- retrieve a list of outstanding back order records sorted by order date
- if no back orders exist, terminate process

Solution for Concern 4

- retrieve the back order record with the oldest date (the next back order in line)
- change status of order from “back order” to “completed”
- save revised order record

Upon assessing the suitability for these actions to be classified as reusable (and forming the basis of entity or utility-centric agnostic functional contexts), the Cutit team reorganizes the logic into different groupings (Figure 11.15) and comes up with the following preliminary agnostic service contexts:

Inventory Processing

- create a new inventory record
- enter chain information, including the assignment of pre-defined model number
- save new inventory record

Chain Information

- retrieve chain specification information by serial number

Order Processing

- issue a query for back orders keyed on model number
- retrieve a list of outstanding back order records (if any) sorted by order date
- retrieve the back order record with the oldest date (the next back order in line)
- change status of order from “back order” to “completed”
- save revised order record

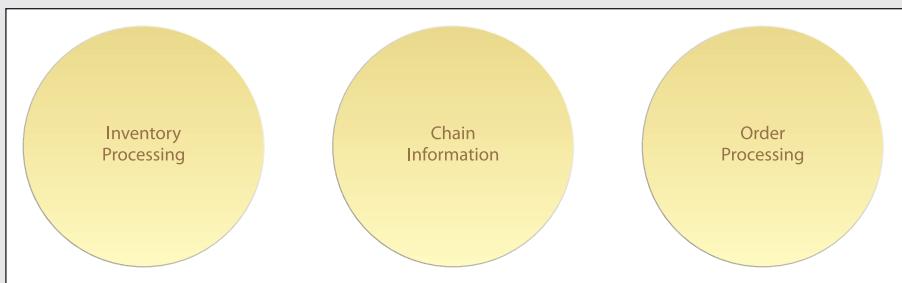
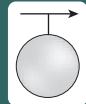


Figure 11.15

The agnostic service contexts established as a result of applying this pattern; all three services are based on Entity Abstraction (175).

Non-Agnostic Context

How can single-purpose service logic be positioned as an effective enterprise resource?



Problem	Non-agnostic logic that is not service-oriented can inhibit the effectiveness of service compositions that utilize agnostic services.
Solution	Non-agnostic solution logic suitable for service encapsulation can be located within services that reside as official members of a service inventory.
Application	A single-purpose functional service context is defined.
Impacts	Although they are not expected to provide reuse potential, non-agnostic services are still subject to the rigor of service-orientation.
Principles	Standardized Service Contract, Service Composability
Architecture	Service

Table 11.4

Profile summary for the Non-Agnostic Context pattern.

Problem

When applying service-orientation, there is a great deal of emphasis on abstracting and positioning solution logic that is agnostic to business tasks and parent business processes. This forms the very basis of the Service Reusability principle and associated patterns.

The result is that non-agnostic logic gets filtered out and often relegated to encapsulation within software programs that are not part of the service inventory but instead exist peripherally as dedicated service consumers (also referred to as “composition initiators”). This is represented by the top part of Figure 11.16.

In this case, service-orientation is not applied to non-agnostic solution logic, which limits its potential to ever become an effective enterprise resource, which can compromise the quality of the service compositions the logic may be responsible for controlling.

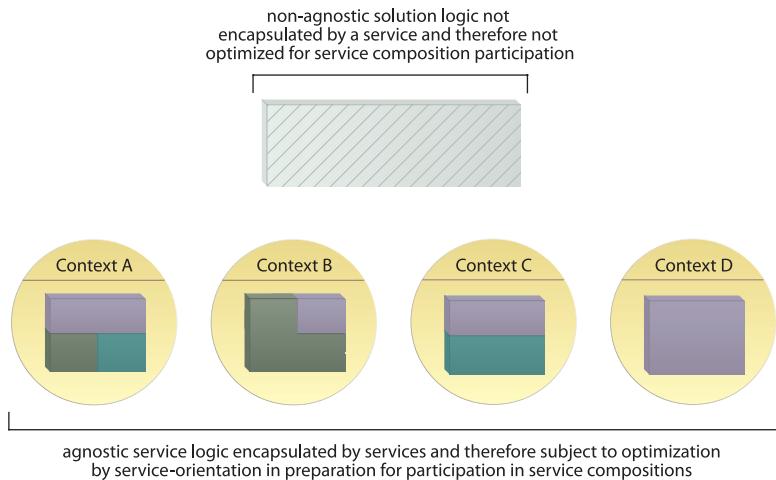


Figure 11.16

The non-agnostic solution logic is not encapsulated into a service and therefore may reduce the effectiveness of service compositions that may include the agnostic services at the bottom of this figure.

Solution

Suitable non-agnostic solution logic is encapsulated by a service with a correspondingly non-agnostic functional context (Figure 11.17). This positions the logic as part of a service inventory. A secondary benefit is that, as a service, this logic is further available for any potential unforeseen involvement in service compositions.

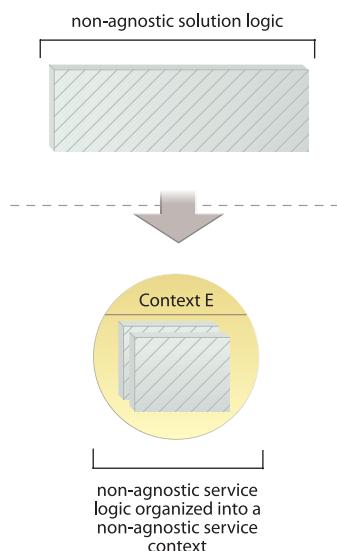


Figure 11.17

The non-agnostic service logic is encapsulated within a service based on a correspondingly non-agnostic service context (E).

Application

Non-agnostic service logic is shaped via the same governing design principles as agnostic services with the exception of Service Reusability and with a lesser initial emphasis on service contract design.

NOTE

If reusable functionality is discovered within the boundary of a non-agnostic service, it can be made available via Agnostic Sub-Controller (607).

This pattern is most commonly applied in combination with Process Abstraction (182) to establish a standard task service layer. However, it is not limited to encapsulating parent business process logic. Other custom, single-purpose service models can be created and based on a non-agnostic functional context.

There are no rules as to whether this pattern should be applied before or after Agnostic Context (312). The mainstream service modeling process described at SOAMethodology.com suggests identifying agnostic service candidates prior to non-agnostic candidates so that multi-purpose logic can be filtered out first, but it is really up to your preferences and whatever methodology you end up using.

Either way, the end result of completing both Agnostic Context (312) and Non-Agnostic Context is that all of the solution logic considered suitable for service encapsulation ends up organized into a set of well-defined service contexts (Figure 11.18).

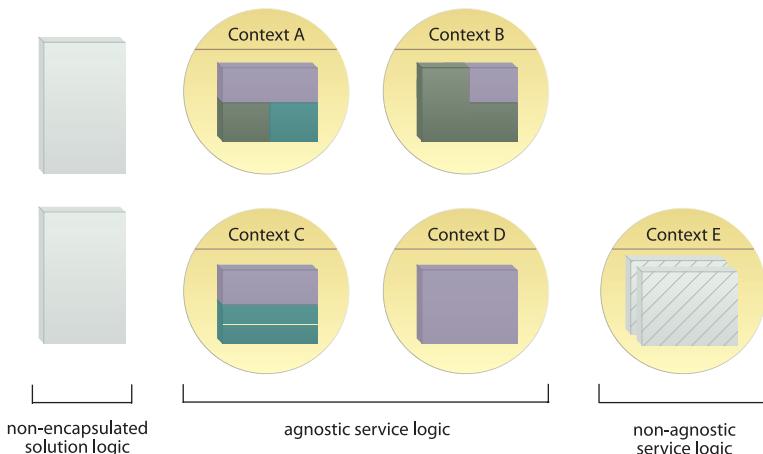


Figure 11.18

Context E joins A through D as future members of a service inventory. The non-encapsulated logic (left) remains separated.

NOTE

As explained in the description for Process Abstraction (182), non-agnostic logic is not required to reside in a service. The assumption when applying this pattern is that the logic allocated for a non-agnostic service was identified as being suitable for encapsulation as per the prior application of Service Encapsulation (305). When applying Non-Agnostic Context as part of a service modeling process, logic designated for a non-agnostic service can still be relocated to a non-service-oriented program. Herbjörn Wilhelmsen published a study at SOAMag.com that compares the pros and cons of abstracting single-purpose logic into services and non-service-oriented applications.

Impacts

Because service-orientation still needs to be applied to the underlying solution logic of a non-agnostic service, its initial delivery will be more expensive and more time-consuming than if it were to simply exist in a program external to the service inventory. The ultimate return on this investment can therefore be significantly lower than with agnostic services.

NOTE

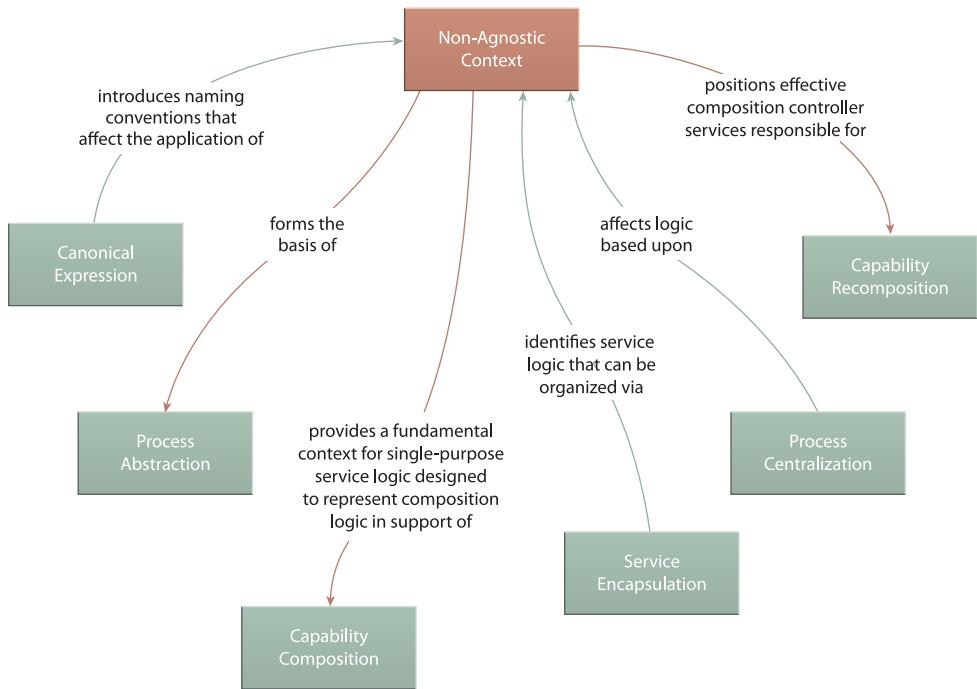
It is the application of this pattern to a body of non-agnostic logic that determines whether this logic is considered a composition initiator or a composition controller. The former is a non-service-oriented program generally responsible for triggering composition logic, whereas the latter is a service responsible for encapsulating composition logic.

Relationships

When studying Non-Agnostic Context, it is important to remember that it is applied subsequent to Service Encapsulation (305). Even though the context is specific to one purpose, it is still considered a service.

The types of services that most commonly require this pattern are those based on task-centric service models. This explains the relationships between Process Abstraction (182) and Process Centralization (193), which are associated with the task service and orchestrated task service models respectively.

A key relationship also defined in Figure 11.19 is that between Non-Agnostic Context and Capability Composition (521). The single-purpose nature of the logic encapsulated by services based on non-agnostic contexts is generally associated with composition logic required to automate a business task. Therefore, this pattern fully supports and even enables Capability Composition (521) and Capability Recomposition (526).

**Figure 11.19**

Non-Agnostic Context establishes a service context that is intentionally single-purpose and very much related to patterns that address parent process design issues.

CASE STUDY EXAMPLE

The definition of a non-agnostic service context for the Chain Inventory Transfer process is quite straightforward primarily because the process is so simple. A parent service context is created to represent the process itself and to assume the responsibility of executing the workflow logic described previously in Figure 11.2.

As shown in Figure 11.20, because this service context is defined by the scope of the business process, it is accordingly named.

**Figure 11.20**

An intentionally non-agnostic service context representing the parent business process is defined. In the Chapter 17 case study examples, this service will be responsible for composing the agnostic services defined earlier in the example for Agnostic Context (312).

Agnostic Capability

How can multi-purpose service logic be made effectively consumable and composable?



Problem	Service capabilities derived from specific concerns may not be useful to multiple service consumers, thereby reducing the reusability potential of the agnostic service.
Solution	Agnostic service logic is partitioned into a set of well-defined capabilities that address common concerns not specific to any one problem. Through subsequent analysis, the agnostic context of capabilities is further refined.
Application	Service capabilities are defined and iteratively refined through proven analysis and modeling processes.
Impacts	The definition of each service capability requires extra up-front analysis and design effort.
Principles	Standardized Service Contract, Service Reusability, Service Composability
Architecture	Service

Table 11.5

Profile summary for the Agnostic Capability pattern.

Problem

When defining service capabilities that were derived from concerns related to a specific problem, there is the natural tendency for those capabilities to be specific to those concerns, regardless of the fact that they reside within an agnostic service context.

This can result in a set of capabilities that may appear to be agnostic but actually provide functionality that is very specific to the concerns associated with the original large problem (or business process) for which they were originally defined (Figure 11.21).

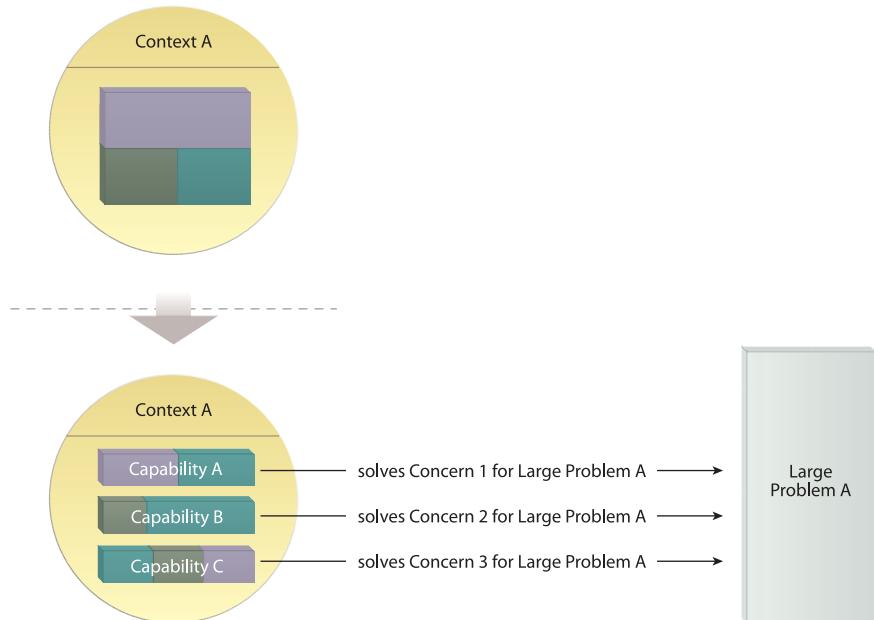


Figure 11.21

The three capabilities provided by Service Context A are defined to solve the specific requirements of the corresponding concerns of Large Problem A. This can reduce and perhaps even eliminate reuse opportunities.

Solution

Agnostic service capabilities are defined and each is subjected to additional analysis beyond its initial definition, allowing it to be refined until it reaches a point where it is sufficiently balanced so that it remains aligned to the parent service's agnostic context while also accommodating the functional requirements of a range of common service consumers. This enables each service capability to address a concern that is truly common (multi-purpose), thereby allowing it to help solve multiple larger problems (Figure 11.22).

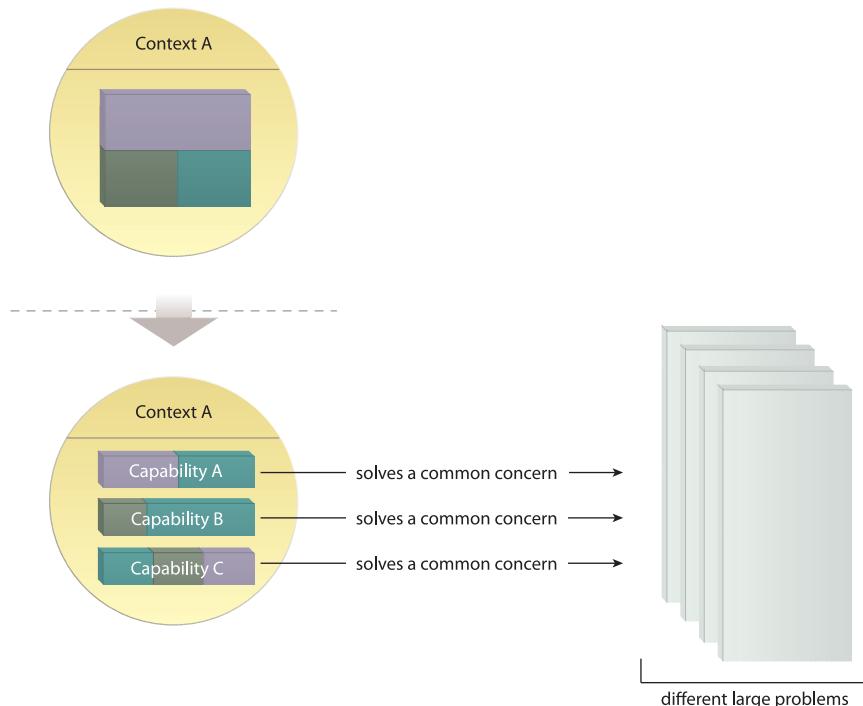


Figure 11.22

Through the application of this pattern, the service logic grouped within a specific service context is made available as a set of well-defined and complementary capabilities.

Application

By carrying out service-oriented analysis and service modeling processes, candidate service capabilities are identified, defined, and grouped into candidate service contexts. Through repeated iterations of these processes, the definition and organization of the capabilities are further refined.

This pattern essentially positions each capability as an independent function able to solve a concern that is common to multiple business processes or tasks. Well-defined agnostic capabilities lie at the heart of fundamental service-orientation principles, such as Service Reusability and Service Composability.

Impacts

The quality of a service capability definition improves with each iteration through a service-oriented analysis process, whereby its functionality and expression (via the service contract) are repeatedly validated or refined. However, all of these iterations add to the up-front analysis time and effort required to produce the service.

Additionally, inadvertent “over modeling” can lead to capabilities that are too vague and too generic or that perhaps offer more functionality than will actually be required. These consequences can be avoided by sticking to analysis processes that are focused on specific business domains.

Relationships

Agnostic Capability can be considered a continuation of Agnostic Context (312), making these two patterns naturally related. But when studying how services are assembled into compositions, the ultimate role of the defined agnostic capabilities becomes evidently integral to the application of both Capability Composition (521) and Capability Recomposition (526).

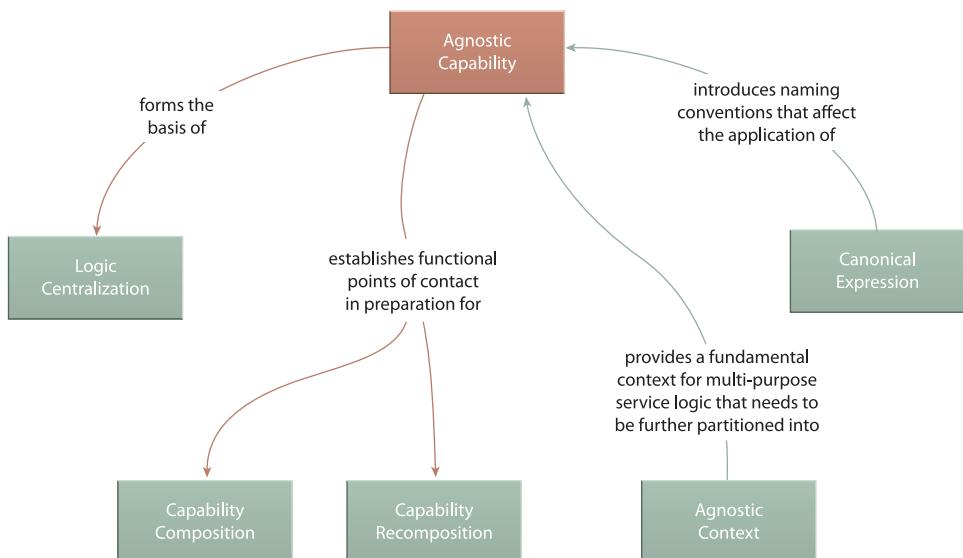


Figure 11.23

Agnostic Capability provides the externally facing functions that form the basis of service contracts.

CASE STUDY EXAMPLE

The Cutit team proceeds to establish service capabilities for its three entity services (Figure 11.24), as follows:

- The actions associated with the Inventory Processing context are studied and consolidated into a single capability called “Create Record.”
- The one action allocated to the Chain Information context is shaped into a generic capability called “Retrieve Chain Record.”
- The actions grouped within the Order Processing context are refined into three separate capabilities called “Retrieve Order Record,” “Retrieve Back Order List,” and “Edit Order.”



Figure 11.24

The three service definitions, each with capabilities that address the processing requirements of the Chain Inventory Transfer business process.

Although there is an opportunity to proceed into the actual design and development stages with these service definitions, the team insists that they perform some additional service modeling. Several new business processes are analyzed, allowing the functionality (and associated algorithms) behind each capability to be better optimized. Another side-benefit of these additional iterations is the streamlining of the service definitions themselves, as shown in Figure 11.25.

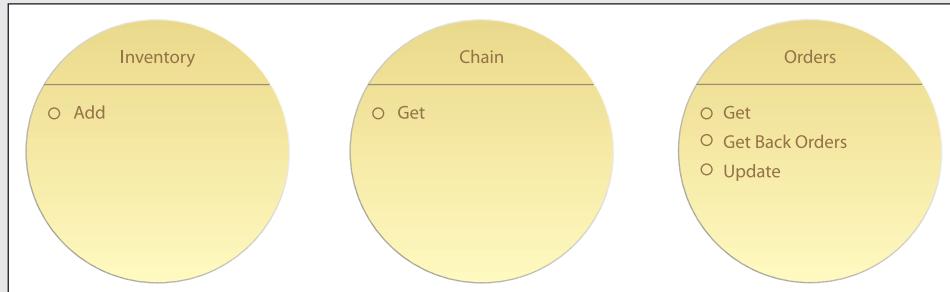


Figure 11.25

After some further service modeling, the definitions are refined with agnostic capabilities.

This page intentionally left blank

Chapter 12



Service Implementation Patterns

Service Façade

Redundant Implementation

Service Data Replication

Partial State Deferral

Partial Validation

UI Mediator

Each of the following design patterns impacts or augments the service architecture in a specific manner, thereby affecting its physical implementation. Most are considered specialized, meaning that they are to be used for specific requirements and may not be needed at all (and are rarely all used together).

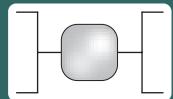
Arguably the most important pattern in this chapter is Service Façade (333) because it introduces a key component into the service architecture that can help a service evolve in response to on-going change.

Redundant Implementation (345), Service Data Replication (350), and Partial State Deferral (356), on the other hand, help establish a more robust service implementation by addressing common performance and scalability demands. Unlike Service Façade (333), which actually alters the complexion of the service architecture, these three patterns can be more easily applied subsequent to a service's initial deployment.

Partial Validation (362) is a consumer-focused pattern that helps optimize runtime message processing and UI Mediator (366) is a pattern specialized to bridging potential usability gaps between services and the presentation layer.

Service Façade

How can a service accommodate changes to its contract or implementation while allowing the core service logic to evolve independently?



Problem	The coupling of the core service logic to contracts and implementation resources can inhibit its evolution and negatively impact service consumers.
Solution	A service façade component is used to abstract a part of the service architecture with negative coupling potential.
Application	A separate façade component is incorporated into the service design.
Impacts	The addition of the façade component introduces design effort and performance overhead.
Principles	Standardized Service Contract, Service Loose Coupling
Architecture	Service

Table 12.1

Profile summary for the Service Façade pattern.

Problem

A given service will contain a core body of logic responsible for carrying out its (usually business-centric) capabilities. When a service is subject to change either due to changes in the contract or in its underlying implementation, this core service logic can find itself extended and augmented to accommodate that change. As a result, the initial bundling of core service logic with contract-specific or implementation-specific processing logic can eventually result in design-time and runtime challenges.

For example:

- A single body of service logic is required to support multiple contracts, thereby introducing new decision logic and requiring the core business routines to process different types of input and output messages.

- The usage patterns of shared resources accessed by the service are changed, resulting in changes to the established service behavior that end up negatively affecting existing service consumers.
- The service implementation is upgraded or refactored, resulting in changes to the core business logic in order to accommodate the new and/or improved implementation.
- The service is subjected to decomposition, as per Service Decomposition (489).

Figure 12.1 illustrates the first of these scenarios.

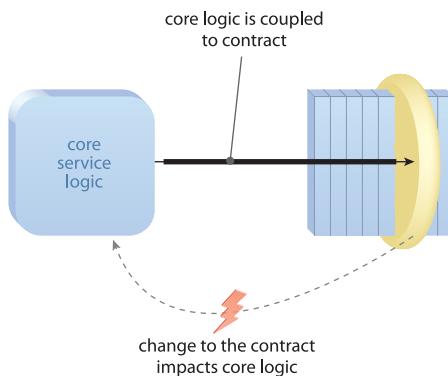


Figure 12.1

If the core service logic is coupled directly to the contract, any changes to how the service interacts with consumers will require changes to the core service logic. (This represents only one of several of the problem scenarios addressed by this pattern.)

Solution

Façade logic is inserted into the service architecture to establish one or more layers of abstraction that can accommodate future changes to the service contract, the service logic, and the underlying service implementation (Figure 12.2).

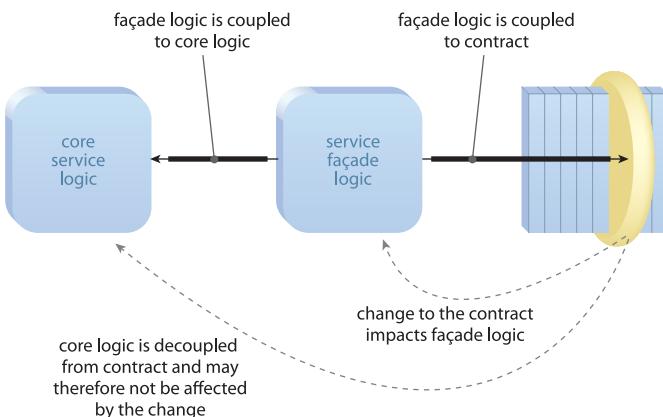


Figure 12.2

Façade logic is placed in between the contract and the core service logic. This allows the core service logic to remain decoupled from the contract.

NOTE

Service Façade is a versatile pattern that can be applied in many different ways within a given service architecture. The upcoming *Application* section explores several possible application scenarios to demonstrate how service façade logic can be potentially utilized. This section is therefore noticeably longer than the average *Application* section for other patterns.

Application

Service façade logic is considered part of the overall service logic but distinct from the core service logic, as follows:

- The core service logic is expected to provide the range of functions responsible for carrying out the capabilities expressed by the service contract.
- The service façade logic is primarily responsible for providing supplemental, intermediate processing logic in support of the core service logic.

Service façade logic is generally isolated into a separate component that is part of the service architecture. Common types of logic that tend to reside within a service façade component include:

- *Relying Logic* – The façade logic simply relays input and output messages between the contract and the core service logic or between the core service logic and other parts of the service architecture. For examples of this, see the descriptions for Proxy Capability (497) and Distributed Capability (510).
- *Broker Logic* – The façade logic carries out transformation logic as per the patterns associated with Service Broker (707). This may be especially required when a single unit of core service logic is used together with multiple service contracts, as per Concurrent Contracts (421).
- *Behavior Correction* – The façade logic is used compensate for changes in the behavior of the core service logic in order to maintain the service behavior to which established consumers have become accustomed.
- *Contract-Specific Requirements* – When service facades are coupled to contracts in order to accommodate different types of service consumers, they can find themselves having to support whatever interaction requirements the contracts express. This can include special security, reliability, and activity management processing requirements. While all of this processing can also be located within the core service logic, it may be desirable to isolate it into façade components when the processing requirements are exclusive to specific contracts.

Service façade components can be positioned within a service architecture in different ways, depending on the nature and extent of abstraction required. For example, a façade component can be located between the core service logic and the contract. Figure 12.3 elaborates on the problem scenario first introduced in Figure 12.1 by showing how a design based on core service logic coupled to the contract can lead to restrictive architectures after multiple contracts enter the picture.

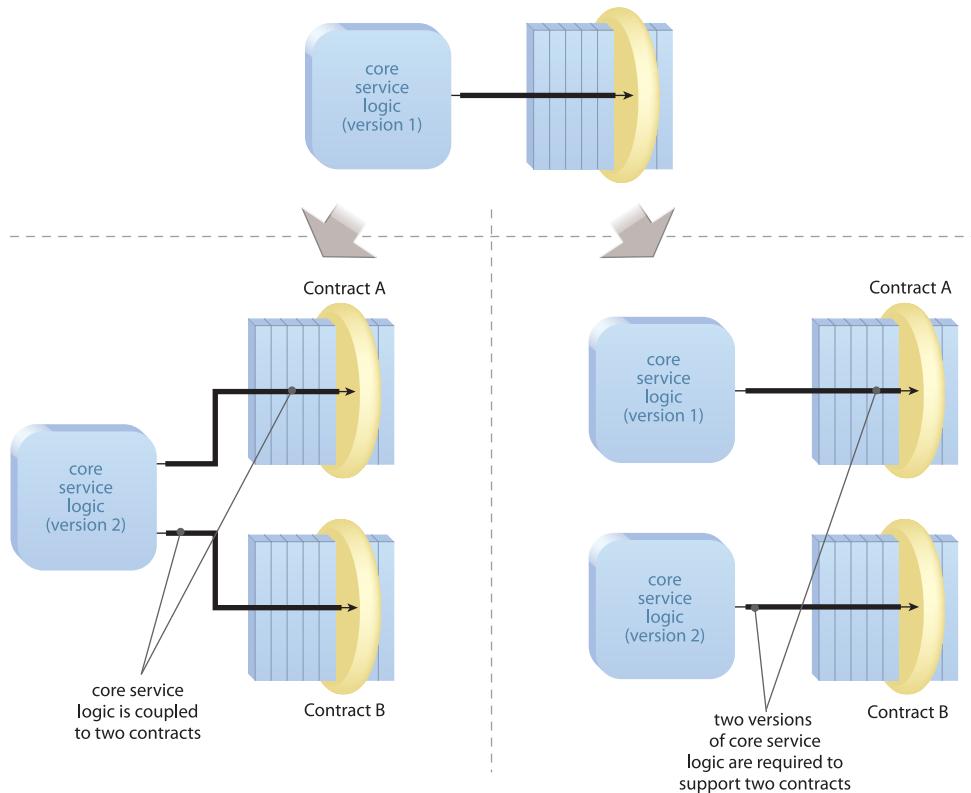
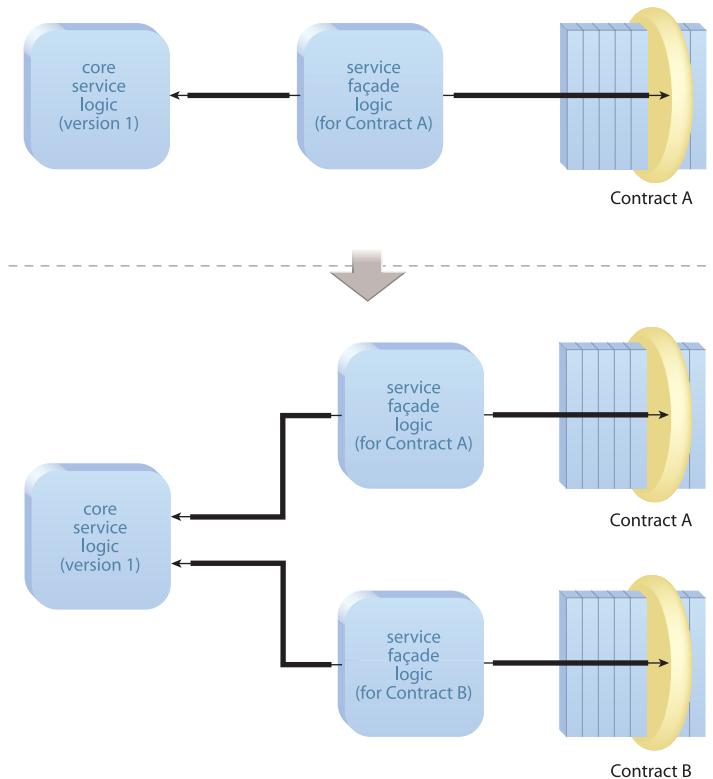


Figure 12.3

When designing services we are encouraged to tailor underlying service logic in support of independently customized and standardized service contracts. This results in a high level of logic-to-contract coupling, while allowing the contract itself to be decoupled from its implementation. Although this is considered desirable from a contract coupling perspective, it can lead to undesirable design options for supporting multiple service contracts with just a base unit of core service logic. The first architecture (left) requires a new version of the core service logic that is now coupled to two contracts, while the second (right) requires the creation of a new service altogether, leading to redundant core service logic.

Figure 12.4 illustrates how the abstraction achieved through the use of service façade components allows for the addition of multiple service contracts without major impact to the core service logic. Service façade components are intentionally tightly coupled to their respective contracts, allowing the core service logic to remain loosely coupled or even decoupled. What this figure also highlights is the opportunity to position consumer-specific service logic as an independent (and perhaps even reusable) part of the service architecture.

**Figure 12.4**

New service contracts can be accommodated by repeating this pattern to introduce new façade components, thereby potentially shielding the core service logic.

When service façade logic is used to correct the behavior of changed core service logic, it is also typically positioned between the contract and core service logic. This allows it to exist independently from logic that is coupled to (and thereby potentially influenced by) specific parts of the underlying implementation.

Figure 12.5 shows how core service logic coupled to both the implementation and the contract may be forced to pass on changes in behavior to established service consumers. Figure 12.6 then demonstrates how a layer of service façade processing can be designed to regulate service-to-consumer interaction in order to preserve the expected service behavior.

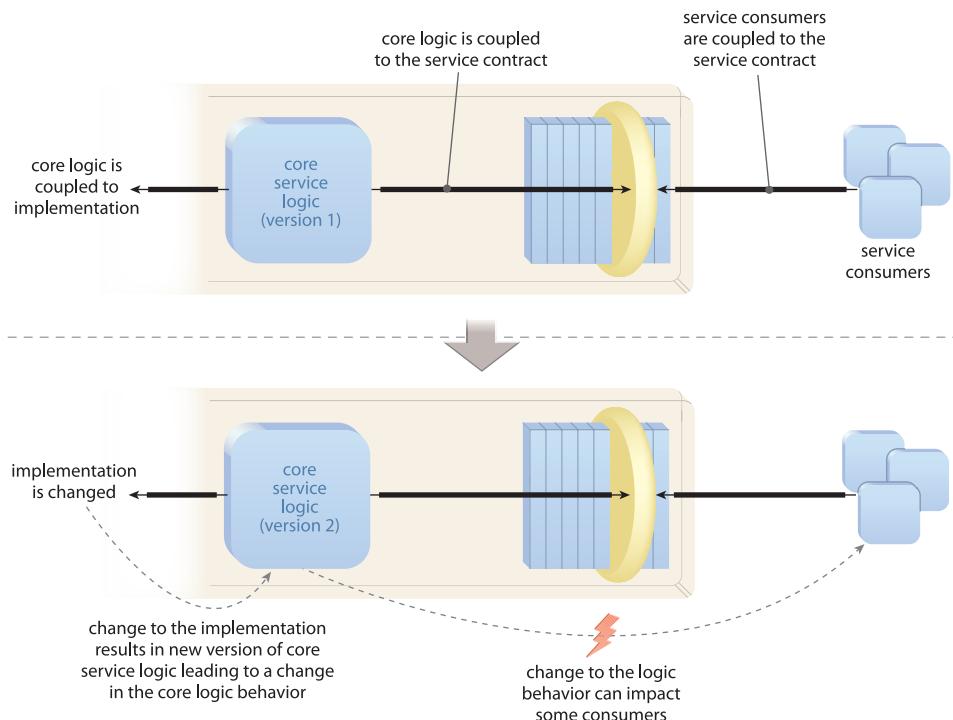


Figure 12.5

Parts of the implementation encapsulated and bound to by version 1 (top) of the core service logic are subject to change, resulting in the release of a second version (bottom) that brings with it a noticeable change in behavior. Service consumers coupled to the service contract are affected by this change because the new core service logic is also directly coupled to the contract.

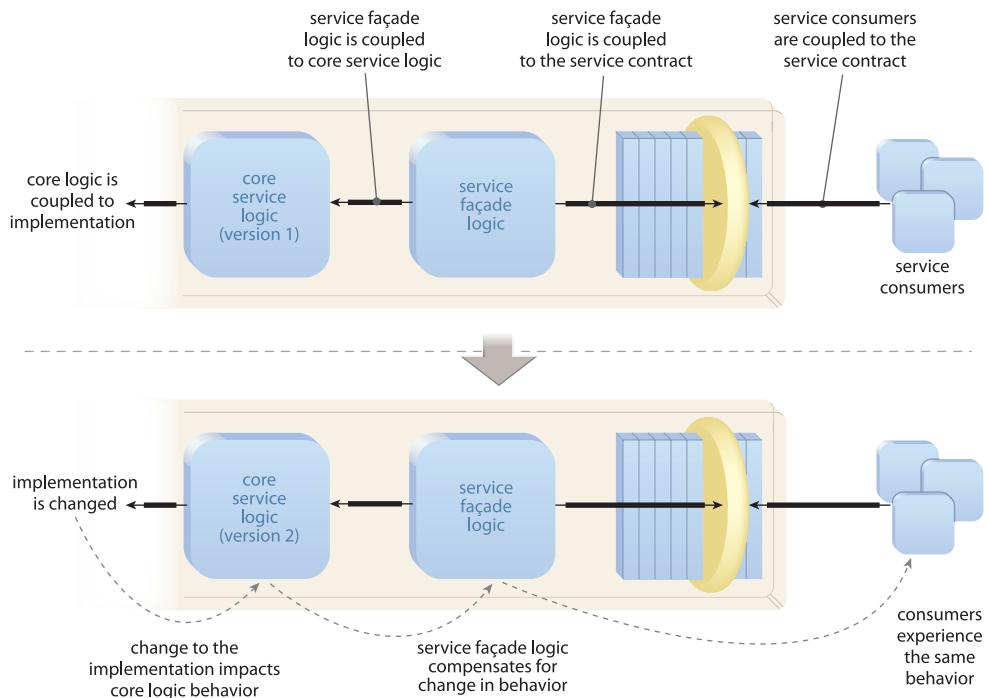


Figure 12.6

The core service logic is updated due to the change in implementation, but the behavioral changes are caught by the service façade component which contains additional routines to preserve the original behavior while still interacting with version 2 of the core service logic.

Finally, it is worth pointing out that service façade logic is not limited to acting as an intermediary between the core service logic and the service contract. Figure 12.7 shows an architecture in which components are positioned as facades for underlying implementation resources. In this case, this pattern helps shield core service logic from changes to the underlying implementation by abstracting backend parts.

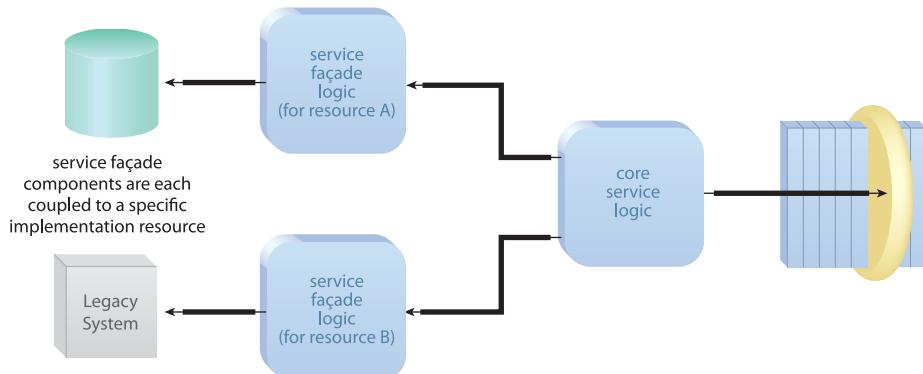


Figure 12.7

One service façade component abstracts a shared database (resource A) whereas another abstracts a legacy system (resource B). This abstraction helps protect the core service logic from changes to either of these parts of the underlying service implementation.

Impacts

Creating façade components results in an increased amount of physical logic decomposition. This naturally introduces additional design and development effort, as well as extra cross-component communication requirements. Although some performance overhead is expected, it is generally minor as long as façade and core service components are located on the same physical server.

Some governance overhead can also be expected, due to the increased amount of components per service.

Relationships

The structural solution provided by Service Façade helps support the application of several other patterns, including Service Refactoring (484), Service Decomposition(489), Proxy Capability (497), Agnostic Sub-Controller (607), Inventory Endpoint(260), Distributed Capability (510), Concurrent Contracts (421), and Contract Denormalization (414). This pattern is ideally combined with Decoupled Contract (401) in order to provide the maximum amount of design and refactoring flexibility throughout a service's lifespan.

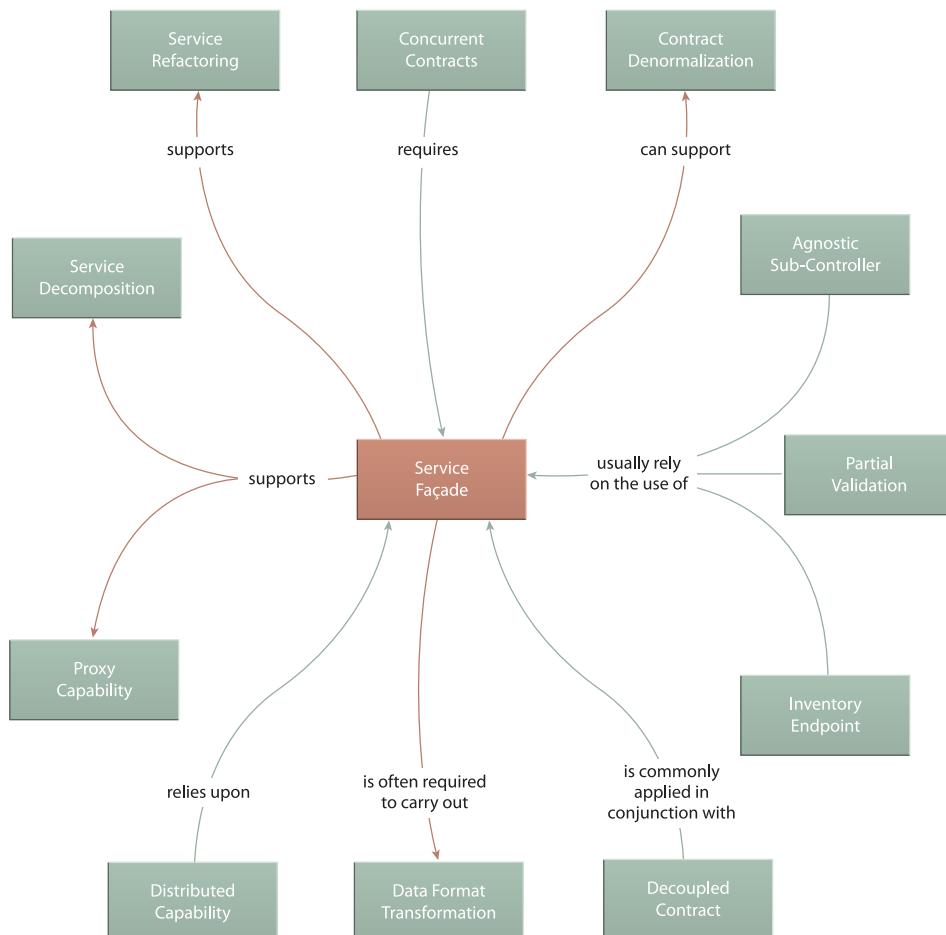


Figure 12.8

Service Façade establishes a key part of the service logic that ends up supporting several other service design patterns.

CASE STUDY EXAMPLE

The FRC is developing an entity service called Appealed Assessments, which is dedicated to producing a range of reports related to already assessed claims that have been successfully or unsuccessfully appealed. Depending on the nature and scope of the requested report, this service may need to access up to six different repositories in order to gather all of the required data.

A component-based architecture already exists in which a separate wrapper utility component has been created to represent and provide standardized access to all six repositories. This Data Controller component provides all the logic required to fulfill the capabilities of the planned Assessment Reports service in addition to several other generic data access and reporting functions.

Instead of creating new logic to accomplish the same data access tasks, FRC wants to use the Data Controller component as the core service logic for the Appealed Assessments service. However, they are told by the group that owns this component that it can't be altered in support of this service. Furthermore, the component is expected to undergo some changes in the near future that may result in it having to support one additional database plus accommodate the planned consolidation of two existing legacy repositories. As a result, the component needs to remain an independently governed part of the architecture.

The FRC architects decide to design a service façade component that will be used to bind to the official WSDL contract for the Appealed Assessments service. The façade component is appropriately named Data Relayer, and its primary responsibility is to receive service consumer requests via the standardized WSDL contract, relay those requests to the corresponding internal wrapper components, and then relay the responses back to the service consumer.

The Data Relayer component contains a modest amount of logic, most of which is focused on validating data reports received from the Data Controller component and (if necessary) converting them to the format and data model required by the Appealed Assessments service's WSDL message construct and associated XML schema complex types.

The resulting service architecture (Figure 12.9) allows the original Data Controller component to evolve independently while establishing the Data Relayer component as an intermediate façade dedicated to the Appealed Assessments service.

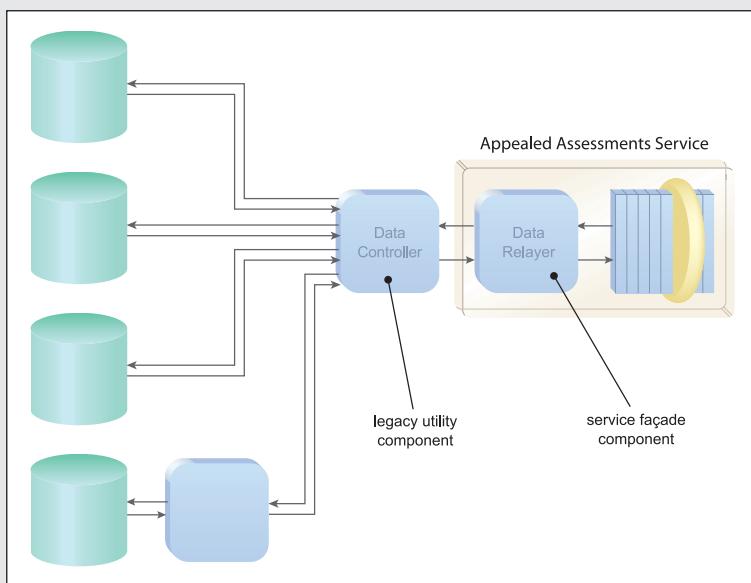


Figure 12.9

The Data Relayer service façade component is designed into the architecture of the Appealed Assessments service. Note the bottom database is accessed via a separate API component. This environment (called “MainAST103”) is explained in the Legacy Wrapper (441) case study example.

This architecture further accommodates the expected changes to the Data Controller component. Should any of these changes affect the format or data of the reports generated by the Data Controller functions, the Data Relayer component can be augmented to compensate for these changes so that the Appealed Assessments service contract remains unchanged and so that consumers of this service remain unaffected.

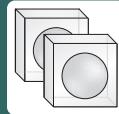
The case study example for Legacy Wrapper (441) continues this scenario by introducing the need to add a legacy wrapper component into the Appealed Assessments service architecture.

NOTE

You may have noticed that in Figure 12.9 Data Controller is further labeled as a “legacy” component. This is because even though it is conceptually similar to a utility service, it is an older component that has not been subjected to service orientation. It therefore is not considered a member of the service inventory but is instead (from an SOA perspective) a part of the legacy environment.

Redundant Implementation

How can the reliability and availability of a service be increased?



Problem	A service that is being actively reused introduces a potential single point of failure that may jeopardize the reliability of all compositions in which it participates if an unexpected error condition occurs.
Solution	Reusable services can be deployed via redundant implementations or with failover support.
Application	The same service implementation is redundantly deployed or supported by infrastructure with redundancy features.
Impacts	Extra governance effort is required to keep all redundant implementations in synch.
Principles	Service Autonomy
Architecture	Service

Table 12.2

Profile summary for the Redundant Implementation pattern.

Problem

Agnostic services are prone to repeated reuse by different service compositions. As a result, each agnostic service can introduce a single point of failure for each composition. Considering the emphasis on repeated reuse within service-orientation, it is easily foreseeable for every complex composition to be comprised of multiple agnostic services that introduce multiple potential points of failure (Figure 12.10).

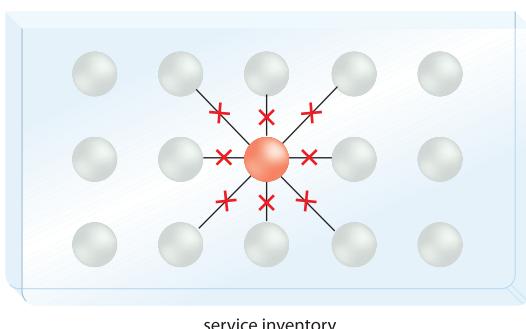


Figure 12.10

When a highly reused service becomes unexpectedly unavailable, it will jeopardize all of its service consumers.

Solution

Multiple implementations of services with high reuse potential or providing critical functionality can be deployed to guarantee high availability and increased reliability, even when unexpected exceptions or outages occur (Figure 12.11).

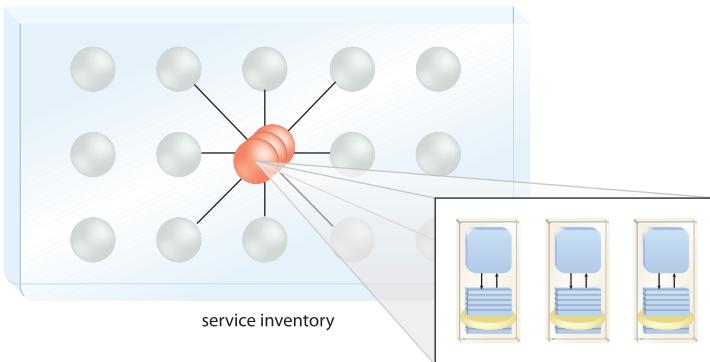


Figure 12.11

Having redundant implementations of agnostic services provides fail-over protection should any one implementation go down.

Application

When services are actually redundantly deployed, there are several ways in which this pattern can be applied:

- Different redundant service implementations can be established for different sets of service consumers.
- One service implementation is designated as the official contact point for consumers, but it is further supported by one or more backup implementations that are used in case of failure or unavailability.

Figure 12.12 illustrates the first variation where the same service is deployed twice; once for access by internal service consumers and again for use by external consumers. This scenario also highlights how this pattern can be applied to various extents. For example, The core service logic may be exactly duplicated in both implementations, but the contracts may, in fact, be different to accommodate the different consumer types, as per Concurrent Contracts (421).

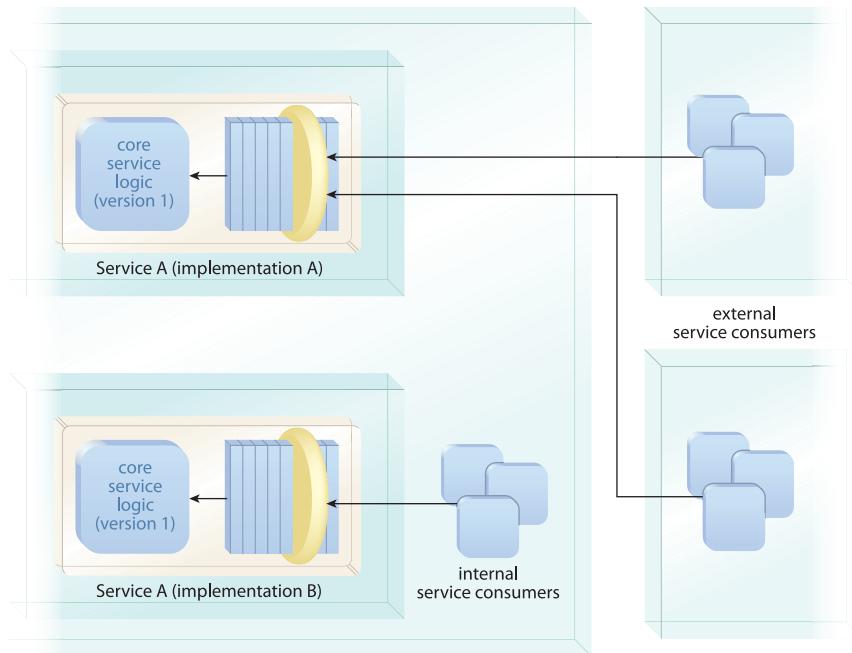


Figure 12.12

Service A has multiple service contracts as well as a redundant implementation, allowing this service to facilitate a wide range of consumer programs.

Impacts

While the application of Redundant Implementation will improve the autonomy, reliability, and scalability of services and the service inventory as a whole, it clearly brings with it some tangible impacts, the foremost of which are increased infrastructure requirements and associated, operational-related governance demands.

For example, additional hardware and administration effort may be needed for each redundantly implemented service and additional governance is required to further keep all duplicated service architectures in sync to whatever extent necessary.

Relationships

Agnostic services naturally have the most concurrent usage demands and therefore have the greatest need for this pattern, which is why it is important for services defined via Entity Abstraction (175) and Utility Abstraction (168). However, even non-agnostic services, such as those realized via Inventory Endpoint (260) may require Redundant Implementation due to reliability demands.

Composition Autonomy (616) will often repeatedly apply Redundant Implementation to ensure that services participating in the composition can achieve increased levels of autonomy and isolation.

Furthermore, establishing a redundant deployment of a service that requires access to shared data sources will usually demand the involvement of Service Data Replication (350).

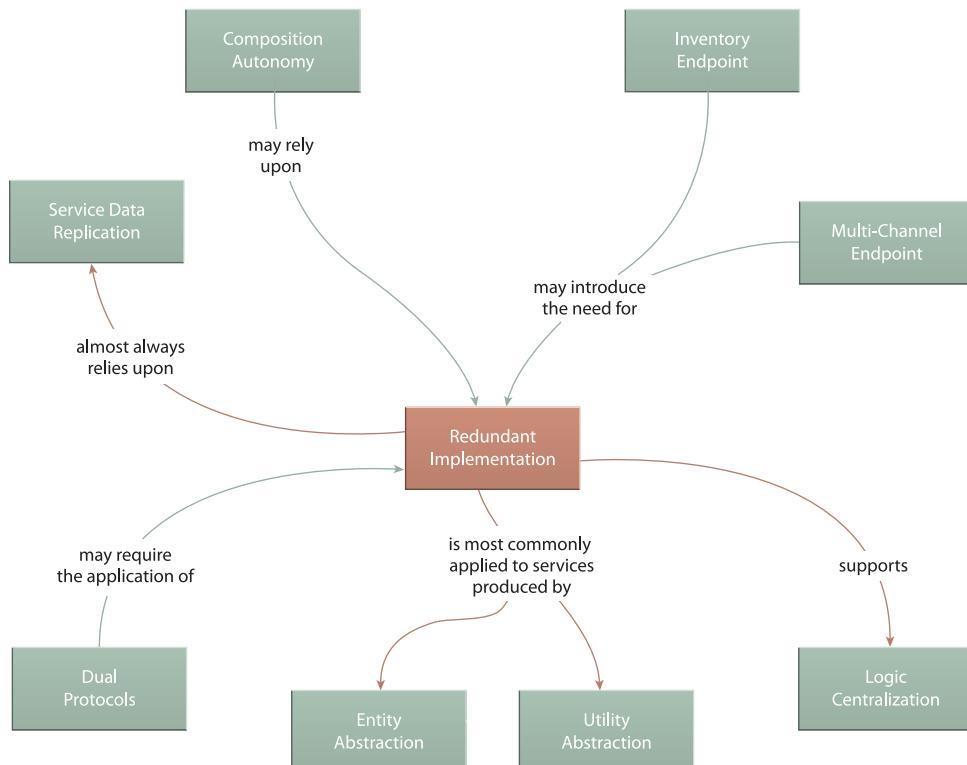


Figure 12.13

Redundant Implementation's support for the Service Autonomy design principle affects several other more specialized (autonomy-related) patterns.

CASE STUDY EXAMPLE

As illustrated in the case study example for Cross-Domain Utility Layer (267), the Alleywood and Tri-Fold service inventories have been architected to share a set of common utility services.

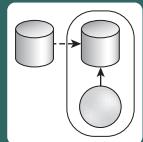
Subsequent to implementing this new cross-domain architecture, some of these utility services naturally became very popular. The Alert service in particular was hit with a consistently high amount of concurrent usage throughout any given work day. Being the service responsible for issuing important notifications when specific pre-defined exception conditions occurred (including policy and security violations), the Alert service was classified as a mission critical part of the overall enterprise architecture.

As a result, a firm requirement was issued, disallowing the Alert service from ever reaching its usage threshold and further requiring chances of service failure be minimized.

To accommodate these requirements, three redundant implementations of the Alert service were created, resulting in four total service implementations. Two were deployed within each environment (Alleywood and Tri-Fold), the second in each environment considered the backup to the first. Intelligent routing agents performed load balancing and failover across each pair of Alert services, as required.

Service Data Replication

How can service autonomy be preserved when services require access to shared data sources?



Problem	Service logic can be deployed in isolation to increase service autonomy, but services continue to lose autonomy when requiring access to shared data sources.
Solution	Services can have their own dedicated databases with replication to shared data sources.
Application	An additional database needs to be provided for the service and one or more replication channels need to be enabled between it and the shared data sources.
Impacts	This pattern results in additional infrastructure cost and demands, and an excess of replication channels can be difficult to manage.
Principles	Service Autonomy
Architecture	Inventory, Service

Table 12.3

Profile summary for the Service Data Replication pattern.

Problem

Various steps can be taken to increase the overall autonomy and behavioral predictability of services. The components that underlie custom-developed services, for example, can be isolated from other programs into their own process space or even onto dedicated servers. These are relatively straightforward measures because the components, the service contract, and even the extra hardware that may be required are all new to the environment.

However, what usually stands in the way of achieving high levels of autonomy is the fact that even the most isolated service will likely still need to interact with some central database in order to access or even update business data. These repositories are usually shared not just with other services, but with various parts of the enterprise, including the legacy applications they may have been originally built for (Figure 12.14).

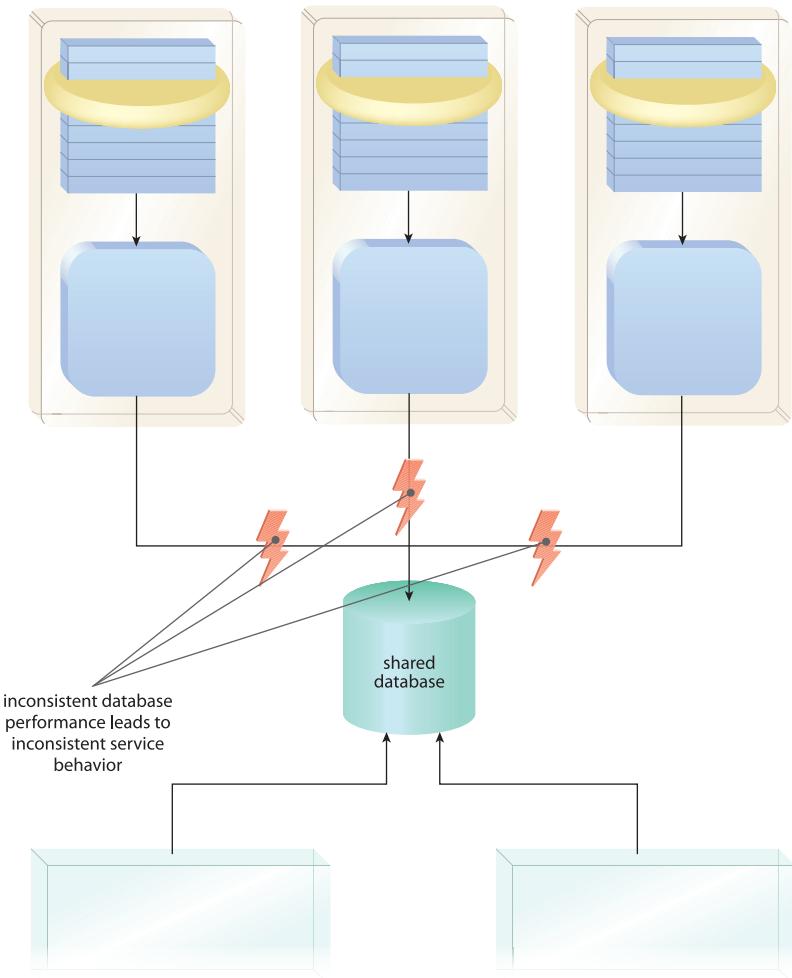


Figure 12.14

Multiple services accessing the same shared database will likely encounter locking and performance constraints that will inhibit their individual autonomy.

Although an organization could choose to rebuild their existing data architecture in support of a new service inventory, the cost, effort, and potential disruption of doing so may be prohibitive.

Solution

Service implementations can be equipped with dedicated databases, but instead of creating dedicated data stores, the databases provide replicated data from a central data source. This way, services can access centralized data with increased autonomy while not requiring exclusive ownership over the data (Figure 12.15).

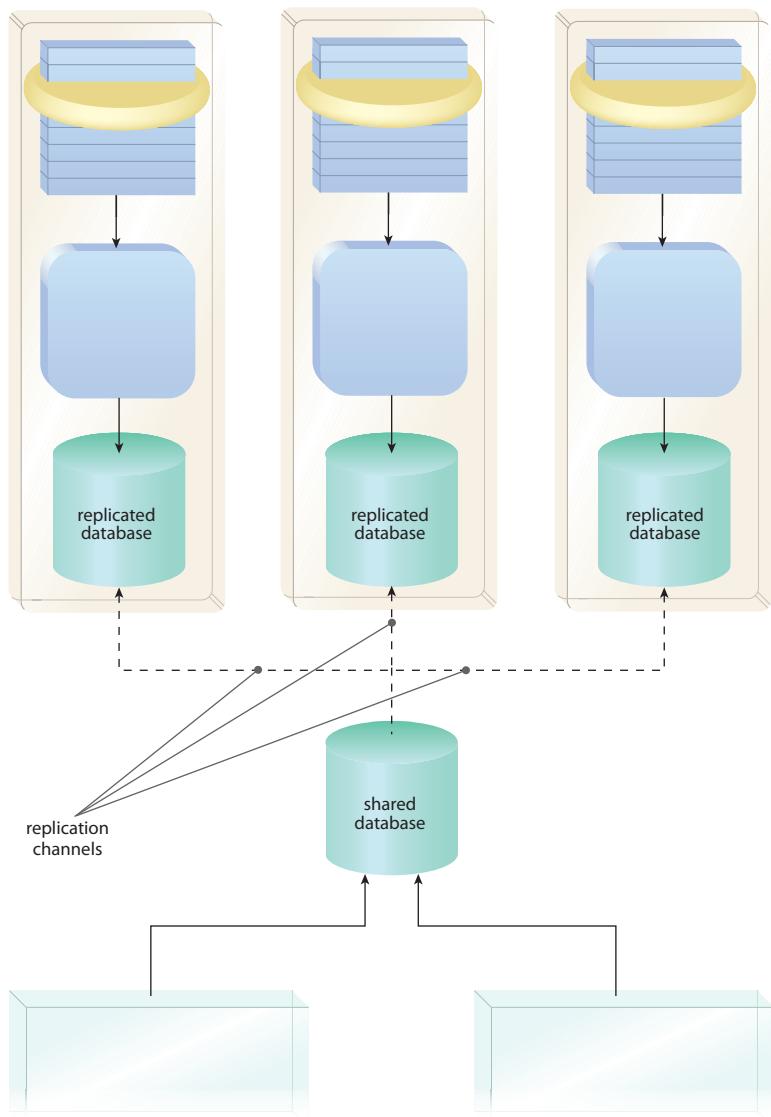


Figure 12.15

By providing each service its own replicated database, autonomy is increased and the strain on the shared central database is also reduced.

Application

This design pattern is especially relevant to agnostic services that are heavily reused and need to facilitate multiple compositions. When this pattern is applied to a large amount of services within a given inventory, it can dramatically reshape the underlying infrastructure of an enterprise's data environment.

Sophisticated data replication architectures may need to be created, and additional design techniques may need to be applied to the databases themselves in order to avoid bottlenecks that can result from an excess of concurrent access and locking. Some replication strategies can even introduce the need for additional satellite databases that provide fully replicated data sets on behalf of a central database but become the contact point for service databases requiring only a subset of the replicated information.

Some services (especially those providing reporting-related capabilities) may only require read access to data, which can be fulfilled by a one-way data replication channel. Most services, though, end up requiring both read and update abilities, which leads to the need for two-way replication.

Furthermore, modern replication technology allows for the runtime transformation of database schemas. As long as the performance and reliability is acceptable, this feature can potentially enable the replicated database to be tuned for individual service architectures.

Impacts

As stated earlier, repeated application of this design pattern can result in costly extensions to the infrastructure in order to support the required data replication in addition to costs associated with all of the additional licenses required for the dedicated service databases. Furthermore, in order to support numerous two-way data replication channels, an enterprise may need to implement a sophisticated and complex data replication architecture, which may require the need to introduce additional, intermediate databases.

Relationships

Service Data Replication is a key pattern applied in support of realizing the Service Autonomy design principle. It ties directly into the application of Redundant Implementation (345) and Composition Autonomy (616) because both aim to reduce service access to shared resources and increase service isolation levels. To access or manage replicated data may further involve some form of legacy interface, as per Legacy Wrapper (441).

Data replication can also play a role in service versioning and decomposition. As shown in Figure 12.11, a replicated data source may be required to support isolated capability logic as defined via Distributed Capability (510), or it may be needed to support already decomposed capability logic resulting from Proxy Capability (497).

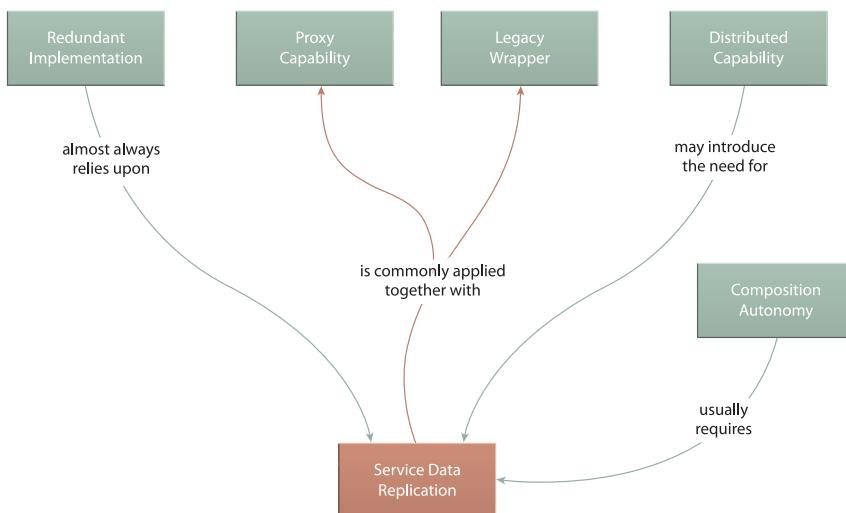


Figure 12.16

Service Data Replication helps reduce the requirements for shared data access and therefore supports a series of autonomy-related patterns.

CASE STUDY EXAMPLE

The FRC Assessment Reports service is responsible for generating and dispensing historical reports for specific registered companies. These reports are used to evaluate and determine annual fines and registration fees (which increase based on the number of violations).

To carry out its reporting functions, this service is required to query the following four databases:

- Registrant Contact (primarily provides company profile information)
- Registrant Activity (includes all incidents, violations, appeals, payments, and other types of historical data)
- Assessment Fees (contains past and current fee schedules and related information)
- Assessment Activity (consists entirely of historical assessments data)

Based on existing design standards that enforce Logic Centralization (136), the first three repositories need to be accessed via the Registrant and Assessment services.

Except for the Assessment Fees database, all of the repositories are used by other legacy applications within the FRC enterprise and now also by other services. Therefore, report generation times can fluctuate, depending on how much shared access is occurring when the Reports service is issuing its queries.

Recently, a new business requirement came about whereby field agents for the FRC would be able to perform assessments on-site while visiting and meeting with registered companies. To perform this task remotely introduced the need for field staff to use portable tablet devices capable of issuing the queries.

To accommodate remote access (especially in regions with limited connectivity) and the increased usage imposed by the new field agent user group, it was decided to improve the response times of assessment report generation by establishing dedicated databases for the Registrant and Assessment Reports services (Figure 12.12).

These databases would be entirely comprised of data replicated from the Registrant Contact, Registrant Activity, and Assessment Activity repositories. Only the subset of data actually required for the reports was replicated and refreshed on a regular basis. The result was a significant increase in autonomy for both Registrant and Assessment Reports services, allowing report generation to be delivered more consistently.

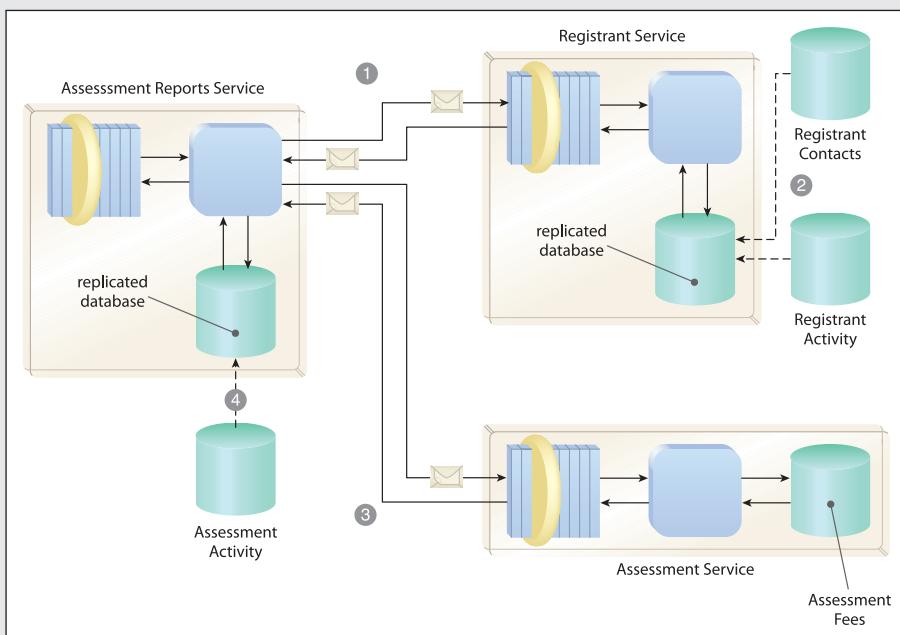
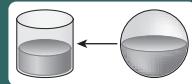


Figure 12.17

The Assessment Reports service first invokes the Registrant service to request Registrant profile and activity data (1). The Registrant Service retrieves this data via a dedicated database comprised of data replicated from the Registrant Contact and Registrant Activity repositories (2). Next, the Assessment Reports service requests assessment fee data via the Assessment service (3). This service already has a dedicated database that does not require replication. Finally, the Assessment Reports service retrieves data from its own replicated Assessment Activity database (4).

Partial State Deferral

How can services be designed to optimize resource consumption while still remaining stateful?



Problem	Service capabilities may be required to store and manage large amounts of state data, resulting in increased memory consumption and reduced scalability.
Solution	Even when services are required to remain stateful, a subset of their state data can be temporarily deferred.
Application	Various state management deferral options exist, depending on the surrounding architecture.
Impacts	Partial state management deferral can add to design complexity and bind a service to the architecture.
Principles	Service Statelessness
Architecture	Inventory, Service

Table 12.4

Profile summary for the Partial State Deferral pattern.

Problem

When services are composed as part of larger runtime activities, there is often a firm need for the service to remain active and stateful while other parts of the activity are being completed.

If the service is required to hold larger amounts of state data, the state management requirements can result in a significant performance drain on the underlying implementation environment. This can be wasteful when only a subset of the data is actually required for the service to accommodate the activity.

In high concurrency scenarios environments, the actual availability of the service can be compromised where accumulated, wasted resources compound to exceed system thresholds (Figure 12.18).

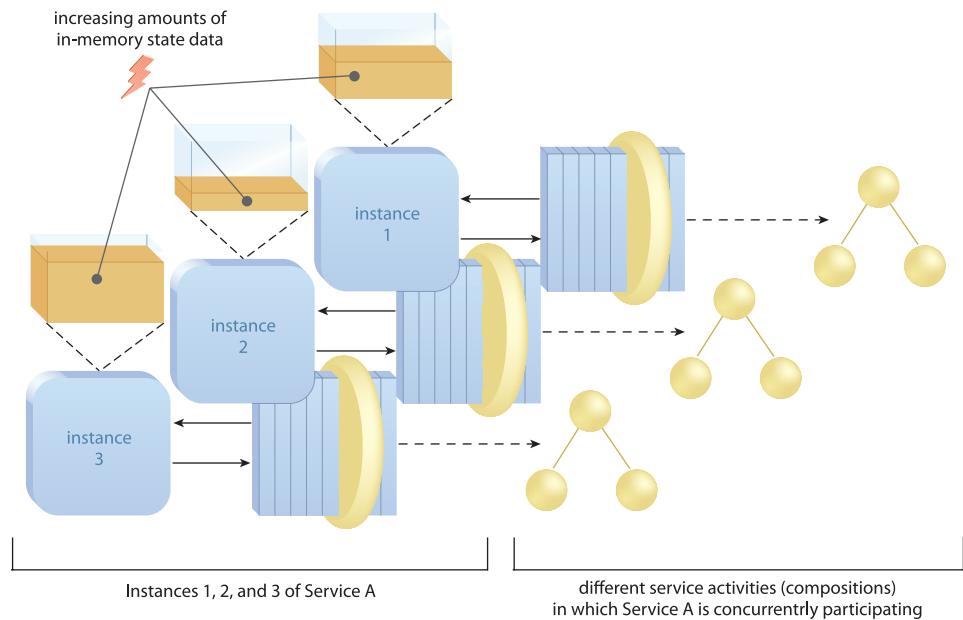


Figure 12.18

In concurrent usage scenarios, stateful services will require that multiple service instances be invoked, each with its own measure of state-related memory consumption requirements.

Solution

The service logic can be designed to defer a *subset* of its state information and management responsibilities to another part of the enterprise. This allows the service to remain stateful while consuming less system resources (Figure 12.19). The deferred state data can be retrieved when required.

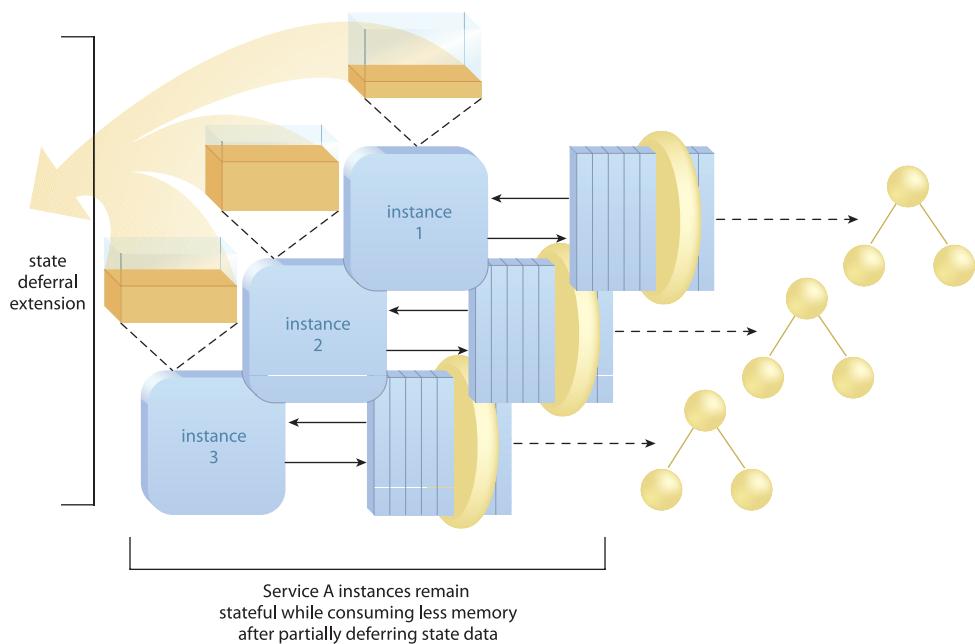


Figure 12.19

Applying this pattern results in the same amount of concurrent service instances but less overall state-related memory consumption.

Application

This design pattern is almost always applied for the deferral of large amounts of business state data, such as record sets or code lists. The general idea is for these bodies of data to be temporarily off-loaded. To accomplish this, an effective state delegation option is required. This may preclude the use of State Repository (242) unless virtual databases can be utilized to make the writing and retrieval of data efficient and responsive.

Partial State Deferral can be effectively used in conjunction with Stateful Services (248) or State Messaging (557) so that state data transmissions can occur without writing to disk. Any state deferral extension can be used in support of this pattern, as long as the performance hit of transferring state data does not introduce unreasonable lag time to the overall activity so that the extension does not undermine the performance gain sought by the pattern itself.

Services designed with this pattern can be further optimized to minimize lag time by retrieving deferred state data in advance.

NOTE

For descriptions of different types of state data and levels of service statelessness, see SOAGlossary.com.

Impacts

Most state management deferral options require that the service move and then later retrieve the state data from outside of its boundary. This can challenge the preference to keep the service as a self-contained part of an inventory and can also bind its implementation to the technology architecture. The resulting architectural dependency may result in governance challenges should standard state management extensions ever need to be changed.

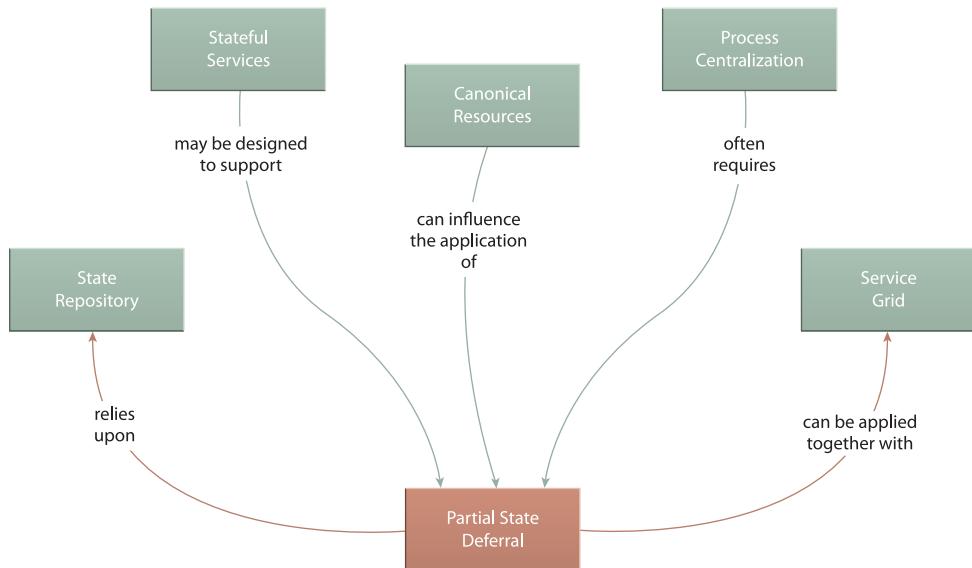
Furthermore, the routines required to program service logic that carries out runtime state data deferral and retrieval add design and development complexity and effort. Finally, if the aforementioned optimization is not possible, the retrieval of large amounts of business data as part of a sequential processing routine will introduce some extent of lag time.

NOTE

The target state sought by this design pattern corresponds to the *Partially Deferred Memory* statelessness level described in Chapter 11 of *SOA Principles of Service Design*.

Relationships

This specialized pattern has relationships with the other state management-related patterns, namely State Repository (242), Service Grid (254), State Messaging (557), and Stateful Services (248), and also provides a common feature used in orchestration environments, as per Process Centralization (193). The application of Canonical Resources (237) can further affect how this pattern is applied.

**Figure 12.20**

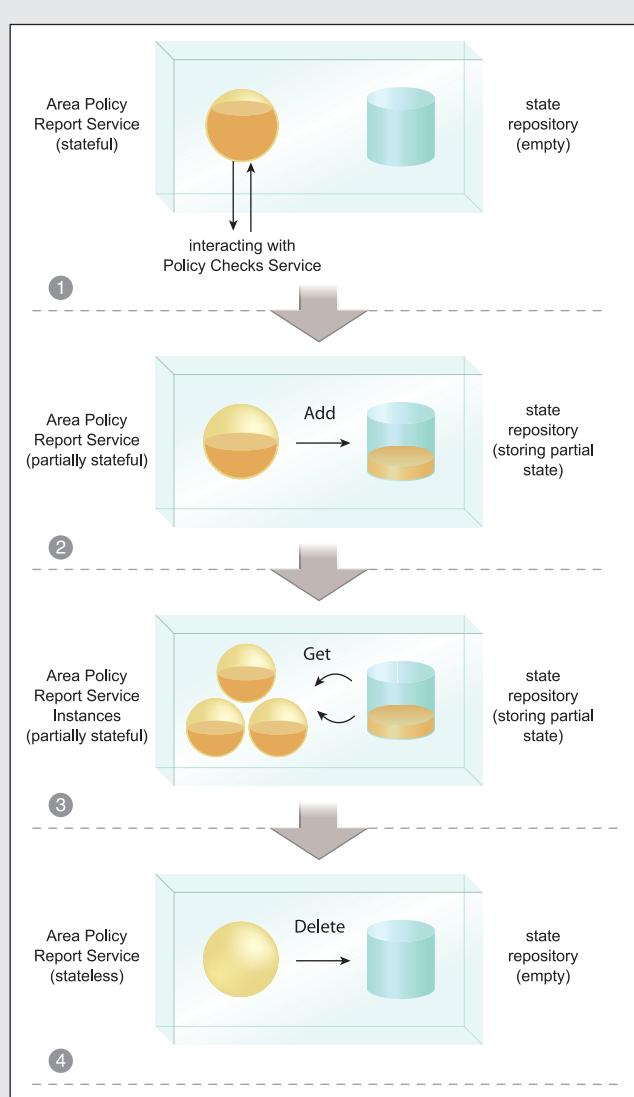
Partial State Deferral has basic relationships with other patterns that support or benefit from state management delegation.

CASE STUDY EXAMPLE

The FRC Area Policy Report service (also described in the Composition Autonomy (616) case study example section) is required to access the Area service and then the Policy Checks service, the latter of which is located on a remote server. Because policy data does not change on a frequent basis and because on any given day most queries issued by the Area Policy Report service are generally related to the same areas, an opportunity is discovered to optimize the composition architecture by applying Partial State Deferral.

Essentially, whenever one or more instances of the Area Policy Report task service are active, the retrieved policy data is stored in a local state repository. Because during the course of a normal working day the majority of reports relate to the same group of areas, the stored policy data is useful to most instances of this service.

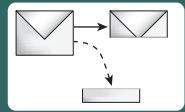
As shown in Figure 12.21, instances of the Area Policy Report task service remain stateful but are not required to explicitly retrieve or store the policy data.

**Figure 12.21**

The first instance of the Area Policy Report service is responsible for retrieving the policy data from the Policy Checks service (1) and populating the state repository (2). Subsequent instances are free to query the state repository (3) instead of accessing the Policy Checks service. Finally, the last instance is responsible for clearing the state repository (4).

Partial Validation

By David Orchard, Chris Riley



How can unnecessary data validation be avoided?

Problem	The generic capabilities provided by agnostic services sometimes result in service contracts that impose unnecessary data and validation upon consumer programs.
Solution	A consumer program can be designed to only validate the relevant subset of the data and ignore the remainder.
Application	The application of this pattern is specific to the technology used for the consumer implementation. For example, with Web services, XPath can be used to filter out unnecessary data prior to validation.
Impacts	Extra design-time effort is required and the additional runtime data filtering-related logic can reduce the processing gains of avoiding unnecessary validation.
Principles	Standardized Service Contract, Service Loose Coupling
Architecture	Composition

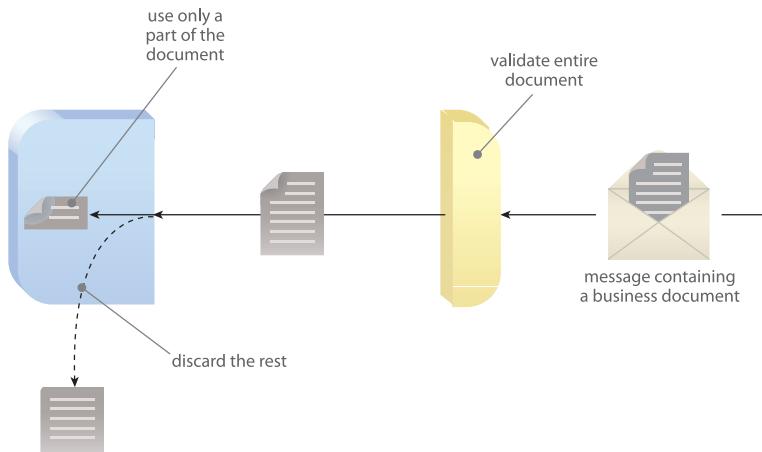
Table 12.5

Profile summary for the Partial Validation pattern.

Problem

Agnostic services are designed with high reuse potential in mind, and therefore there is a constant emphasis on providing generic capabilities that can accommodate a wide range of possible consumers.

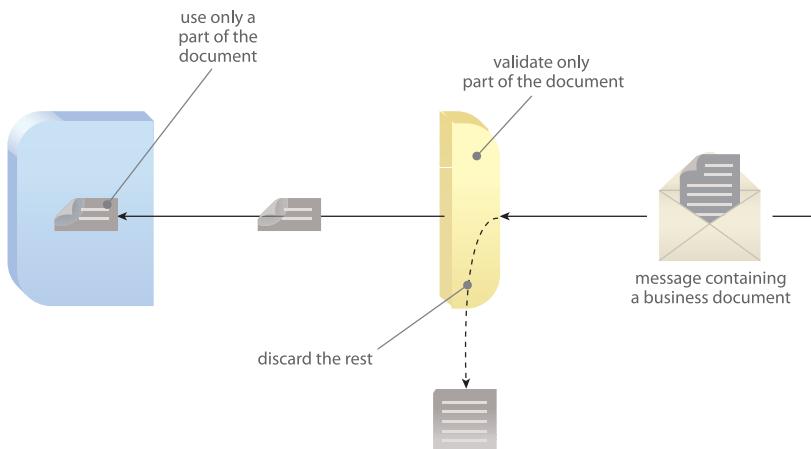
Although this approach leads to increased reuse opportunities, it can also impose unreasonable validation requirements upon some consumers. A typical example is when a capability is designed to be intentionally coarse-grained in order to provide a broad data set in its response messages. The set of data may only be useful to a subset of the service consumers the remaining of which will be forced to validate the message data upon receiving it but then discard data that is not relevant to their needs (Figure 12.22).

**Figure 12.22**

When a service consumer requires only a subset of the data provided to it by the agnostic service, it is expected to validate the entire data set (message payload) before discarding the unnecessary message data.

Solution

The service consumer is intentionally designed to not fully comply to the service contract. Instead, its validation logic is tuned to only look for and validate message data relevant to its needs, thereby ignoring the rest (Figure 12.23). This reduces consumer processing requirements and decreases the extent to which some consumers need to couple themselves to the service contract.

**Figure 12.23**

Because the irrelevant data is ignored prior to validation, it is discarded earlier and avoids imposing unnecessary validation-related processing upon the consumer.

Application

Partial Validation is applied within the consumer program implementation. Custom routines are added to allow for the regular receipt and parsing of incoming service messages, while then avoiding actual validation of irrelevant data.

A typical algorithm used by these routines would be as follows:

1. Receive response message from service.
2. Identify the parts of the message that are relevant to the consumer's processing requirements.
3. Validate the parts identified in Step 2 and discard the balance of the message contents.
4. If valid, retain the parts identified in Step 2. Otherwise, reject the message.

Partial Validation routines can be located within the core consumer business logic or they can be abstracted into an event-driven intermediary, as per Service Agent (543). When services assume the consumer role by composing other services, this type of logic may also be suitable for abstraction via Service Façade (333).

Impacts

The custom programming required by this pattern can add to the design complexity of the overall consumer program logic. Furthermore, the extra processing required by the consumer to look for and extract only relevant data can impose its own processing overhead.

Relationships

Depending on how Partial Validation is applied, it may make sense to combine it with Service Agent (543) or Service Façade (333). Although not directly related to the application of Validation Abstraction (429), these two patterns do share common goals.

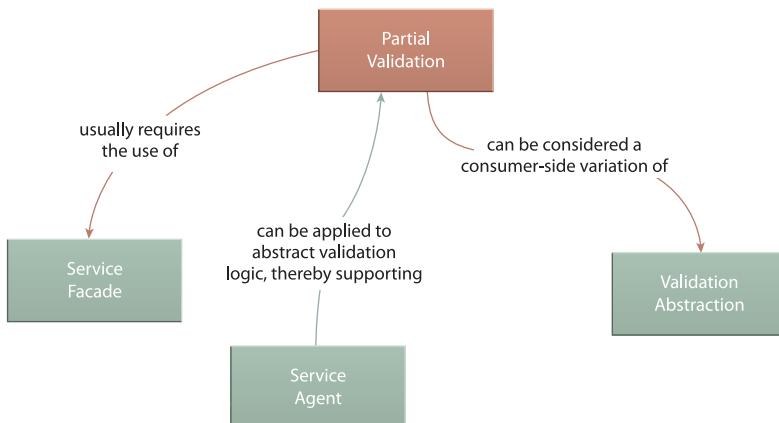


Figure 12.24

Partial Validation introduces internal, consumer-side processing logic and therefore has limited relationships with other patterns.

CASE STUDY EXAMPLE

The FRC Assessment Reports service (described earlier in the Service Data Replication (350) case study example) is required to access the Registrant service in order to request registrant profile data for one of its reports (see Figure 12.12). These reports are often parameter-driven, meaning that they can vary in scope and content depending on the reporting parameters provided to the Assessment Reports service by a given consumer.

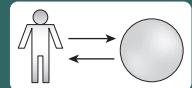
Because of this variance in report content and because the Assessment Reports service is always required to invoke the same Get operation that returns entire profile documents, it often ends up with more registrant data than it actually needs. Its original design simply accepted and validated incoming messages and then made the entire message contents available to the report generation routines. However, as the amount of concurrent usage increases, so does the complexity of some reports, leading to increased resource requirements for this service.

Due to the enterprise-wide initiative to reduce infrastructure costs, architects do not receive funding to apply Redundant Implementation (345) in order to establish a second implementation of this service for load balancing purposes. This forces them to revisit the service design in order to investigate optimization opportunities.

They soon discover that the internal service logic can be refactored by applying Partial Validation. This enables the service to continue performing its report generation logic while decreasing its processing and memory consumption due to a dramatic reduction in runtime message validation.

UI Mediator

By Clemens Utschig-Utschig, Berthold Maier,
Bernd Trops, Hajo Normann, Torsten Winterberg



How can a service-oriented solution provide a consistent, interactive user experience?

Problem	Because the behavior of individual services can vary depending on their design, runtime usage, and the workload required to carry out a given capability, the consistency with which a service-oriented solution can respond to requests originating from a user-interface can fluctuate, leading to a poor user experience.
Solution	Establish mediator logic solely responsible for ensuring timely interaction and feedback with user-interfaces and presentation logic.
Application	A utility mediator service or service agent is positioned as the initial recipient of messages originating from the user-interface. This mediation logic responds in a timely and consistent manner regardless of the behavior of the underlying solution.
Impacts	The mediator logic establishes an additional layer of processing that can add to the required runtime processing.
Principles	Service Loose Coupling
Architecture	Composition

Table 12.6

Profile summary for the UI Mediator pattern.

Problem

Service-oriented solutions are commonly designed as service compositions that may be comprised of services with varying runtime behaviors and processing demands. When the process being automated by the solution is driven by human interaction via user-interfaces, the quality of the user experience can vary due to these behavioral and environmental irregularities.

Whereas a human user expects immediate responses to requests issued via the user-interface, the underlying services may not be designed or able to provide these responses in a synchronous and timely manner (Figure 12.25). Poor or inconsistent user experience can lead to a decrease in the usage and overall success of the solution.

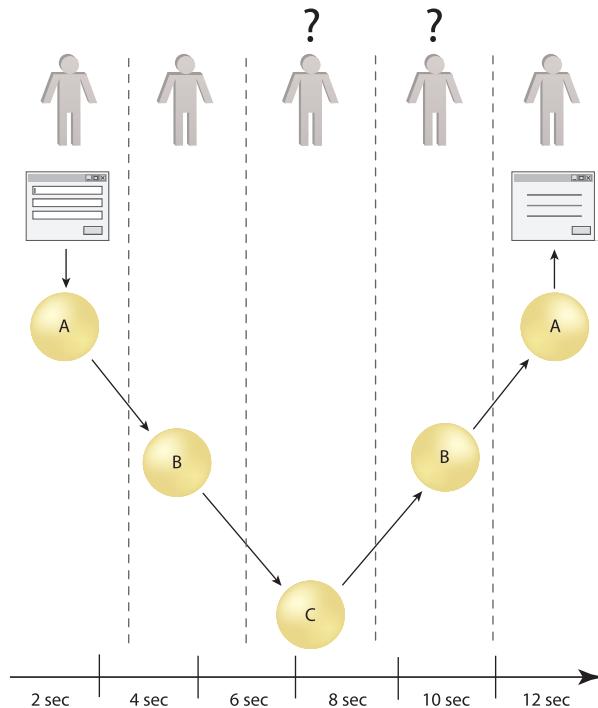


Figure 12.25

While services A, B, and C require several seconds to automate a task initiated via a user-interface, the human user receives no indication as to the progress of the task and is left waiting until a result is finally displayed.

Solution

A mediator service is positioned between a service or service composition and the front-end solution user-interfaces. It is responsible for providing the user with continuous feedback and for gracefully facilitating various runtime conditions so that the underlying processing of the services does not affect the quality of the user experience (Figure 12.26).

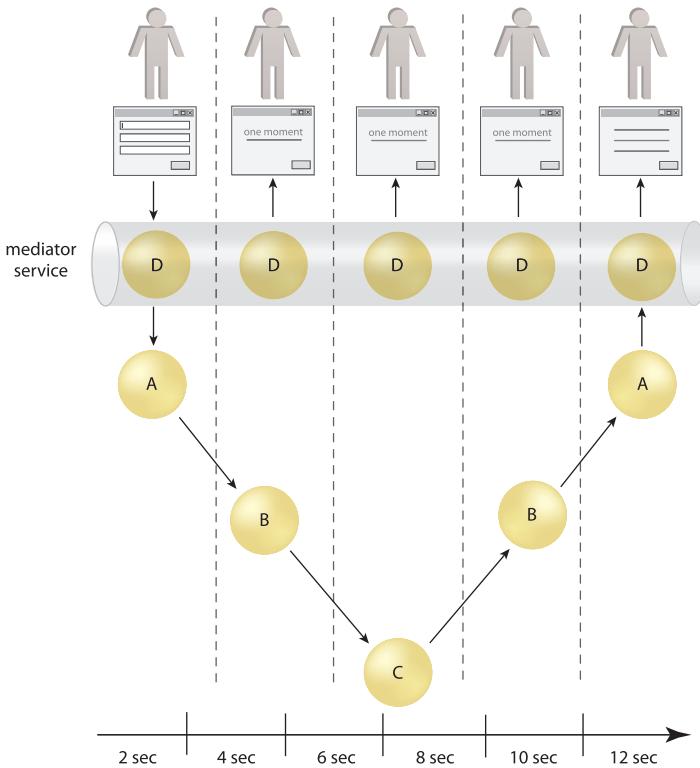


Figure 12.26

The mediator service (D) regularly updates the user interface while services A, B, and C work behind-the-scenes to complete the task.

Application

There are two common methods of applying this pattern:

- Build a mediator service with its own service contract.
- Build a mediator service agent.

The first approach requires that a mediator utility service be created and that the user-interface be designed to bind solely with this service for the duration of a specific task. The mediator service exposes a generic contract with weakly typed capabilities that simply relay request and response messages between the user-interface and underlying service(s). This type of mediator service will contain logic that determines how and when to interact with the user-interface independently.

The second approach requires that this pattern be applied together with Service Agent (S43). When locating mediator logic within event-driven agents, request and response messages between the user-interface and services are transparently intercepted, triggering events that kick-off the mediation logic. Agents must be designed to remain stateful during the completion of the task so that they can interact with the user-interface as required.

Common user-interface mediation routines include:

- displaying forms or pages with a progress or status indicator while services are processing a given request
- displaying forms that request additional data from the user
- routing a user task to the next step independently from underlying service processing
- simulating synchronous human-to-solution exchanges while underlying service activities are carried out asynchronously
- gracefully responding to exception or time-out conditions

The mediator essentially preserves a constant correlation between a user session and the process being automated by services. For this purpose, the mediator service or agent may even maintain a correlation ID that is assigned to all incoming and outgoing messages. However, in order for mediation logic to remain agnostic, it will generally not contain any business process-specific rules or logic. Its capabilities are limited to generic interaction routines.

Impacts

When delivering the mediator logic as a service or a service agent, additional runtime processing is added to the automation of the overall business task due to the insertion of the mediator service layer within the overall composition (and also due to the frequent interaction carried out independently by the mediator logic).

Furthermore, when the mediator exists as a service with its own published contract, user-interfaces are required to bind directly to and interact with the mediator service during the span of an entire business task. This naturally decouples the user-interfaces from the underlying service composition, which can be advantageous if the business logic is subject to change. However, if the mediation logic is agnostic and positioned as part of a reusable utility service, the parent composition logic may actually be responsible for controlling its involvement, leading to a tighter coupling.

Relationships

Because it represents a form of utility logic, UI Mediator is based on Utility Abstraction (168). Service Agent (543) simply provides an optional implementation medium for this pattern.

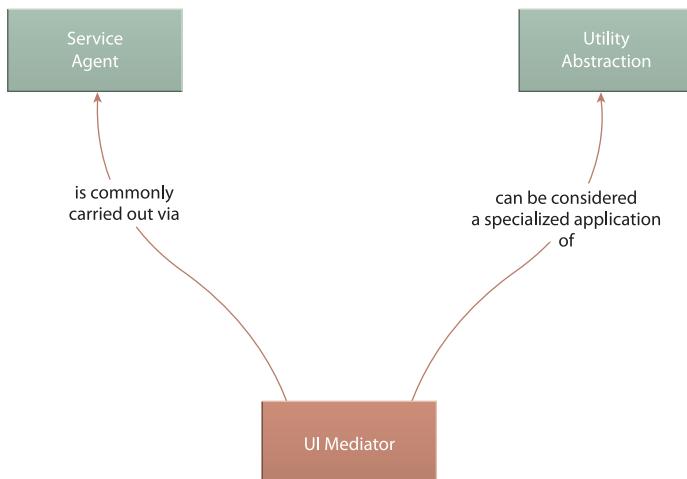


Figure 12.27

As a specialized utility service, UI Mediator has few relationships.

CASE STUDY EXAMPLE

Prior to the purchase of Alleywood Lumber, McPherson had been carrying out an ongoing BPM initiative to help automate existing manual processes and to optimize outdated automated processes. With the assimilation of Alleywood operations, McPherson analysts subject a series of former Alleywood business processes to their established business modeling practices.

The first process relates to the online commercial and retail sale of lumber. Alleywood has a simple Web site in place that allows clients to perform the following tasks sequentially:

1. Initiate a search on the availability of different types of lumber.
2. Assemble an order from the available items.
3. Indicate the shipping details.

The Web site contains scripts that run on the Web server and integrate with an outdated legacy database in which lumber items are stored. Because this database is shared throughout the Alleywood enterprise, its performance and response time can vary dramatically. This primarily affects the time it takes to complete Step 1.

Analysts investigate the history of this commerce site by studying usage logs. After consolidating some of the statistics using a special tool, they discover that the search action can take up to 60 seconds to complete. They further find out that 30% of users who initiate a search abandon the Web site when the query time exceeds 20 seconds, whereas when the query time is less than 5 seconds, 90% of searches result in actual orders.

These metrics are compiled into a report that provides a series of recommendations as to how the online lumber ordering site can be improved, with an emphasis on user experience. This report is passed on to the architecture team, which responds by making a number of changes:

- Service Data Replication (350) is applied to establish a dedicated database for the site.
- A Retail Lumber service is created to provide standardized data access to the database.
- UI Mediator is applied using Service Agent (543) to ensure that long query times or unexpected erratic behavior do not affect the user experience.

The mediator logic contains the following built-in rules:

- If there is no response from the Retail Lumber service within 5 seconds, display a progress indicator page in the user's browser.
- If there is no response from the Retail Lumber service after 15 seconds, display a Web page with a message explaining that the system is currently not available but that the requested information will be e-mailed to the user shortly (the e-mail address is captured as part of the login credentials required to access the site).
- If the Retail Lumber service responds with no data, display the original form along with a message indicating that different search parameters must be provided.

The mediator agent helps establish an improved user experience that appears to be synchronous and interactive, regardless of the behavior of the Retail Lumber service and its underlying database.

This page intentionally left blank

Chapter 13



Service Security Patterns

Exception Shielding

Message Screening

Trusted Subsystem

Service Perimeter Guard

Because service-oriented solutions are typically composed of aggregated services, each moving part within a composition architecture can become a potential target for a security breach. The individual service architectures themselves therefore often need to be equipped with extra controls that enable them to withstand common forms of attacks from malicious consumers.

The following chapter provides four patterns that extend service design in support of increased protection from security threats. Exception Shielding (376) ensures that any error or exception information generated by a service is safe before it is released to consumers. Message Screening (381) is more concerned with inbound data as it provides additional logic that checks messages received from consumers for potentially harmful content. Trusted Subsystem (387) establishes a mechanism whereby consumers cannot directly access service resources with their credentials, and Service Perimeter Guard (394) introduces a new type of utility service that carries out common security functions for external consumers on behalf of internal services.

CASE STUDY BACKGROUND

After Alleywood's reengineered online Lumber Ordering system (described in the case study example for UI Mediator (366) from Chapter 12) is released, McPherson decides to promote it beyond local clients, as part of an international marketing campaign. As a result, the online ordering site attracts many new visitors from foreign regions. Some visit the site just to check out Alleywood's retail inventory, while others find the costs of retail lumber attractive enough to place orders that need to be shipped to remote locations.

However, an unexpected side-effect of the site's increased exposure is an increase in attempted attacks. After just a few days online, the ordering site begins to crash on a daily basis. Days later, the frequency of system failure increases to several times a day. Alleywood architects scramble to find out why this is happening, and they work closely together with McPherson security specialists to investigate the event and usage logs.

What is subsequently revealed is a series of vulnerabilities in the service architecture of the Retail Lumber service. The security team insists that the site and compromised

resources be taken off-line until this issue is resolved. They are concerned that hackers will gain enough environmental information to attack other parts of the enterprise.

As a result of a thorough threat analysis, Alleywood architects propose a site redesign, in which these vulnerabilities in the Retail Lumber service architecture are addressed using well-known service security patterns (as explained in the upcoming examples in this chapter).

Exception Shielding

By Jason Hogg, Don Smith, Fred Chong, Tom Hollander, Wojtek Kozaczynski, Larry Brader, Nelly Delgado, Dwayne Taylor, Lonnie Wall, Paul Slater, Sajjad Nasir Imran, Pablo Cibraro, Ward Cunningham



How can a service prevent the disclosure of information about its internal implementation when an exception occurs?

Problem	Unfiltered exception data output by a service may contain internal implementation details that can compromise the security of the service and its surrounding environment.
Solution	Potentially unsafe exception data is “sanitized” by replacing it with exception data that is safe by design before it is made available to consumers.
Application	This pattern can be applied at design time by reviewing and altering source code or at runtime by adding dynamic sanitization routines.
Impacts	Sanitized exception information can make the tracking of errors more difficult due to the lack of detail provided to consumers.
Principles	Service Abstraction
Architecture	Service

Table 13.1

Profile summary for the Exception Shielding pattern.

Problem

When an exception condition occurs inside a service implementation, the service may issue a response message to convey the exception to the consumer. As shown in Figure 13.1, the response message may inadvertently contain unsafe information that can be exploited to attack the service and its surrounding environment.

For example, a detailed fault message can disclose information about the resources accessed by the service logic that threw the exception. An attacker could then deliberately cause the service to throw an unhandled exception in an attempt to obtain and exploit sensitive information, such as connection strings, server names, SQL queries, XPath commands, stack traces, and data schemas.

Furthermore, if an exception is expected, a pre-defined error message with information about the cause of the fault could be returned to the consumer. Such a message may have been designed by an architect or developer without knowledge of its security implications and may therefore also contain sensitive information that poses a security risk.

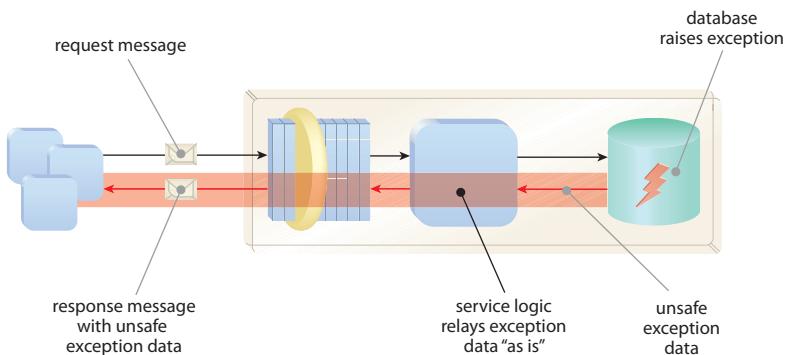


Figure 13.1

Subsequent to an exception, the service logic passes on potentially unsafe information to service consumers.

Solution

Unsafe exception-related data is “sanitized,” a process by which this information is identified and replaced with exception information that is safe by design. Sanitized exception messages do not contain sensitive data nor a detailed stack trace, either of which might reveal potentially harmful details about the service’s inner workings. After a service is subjected to a sanitization process, it is limited to returning only those exception details that are deemed safe (Figure 13.2).

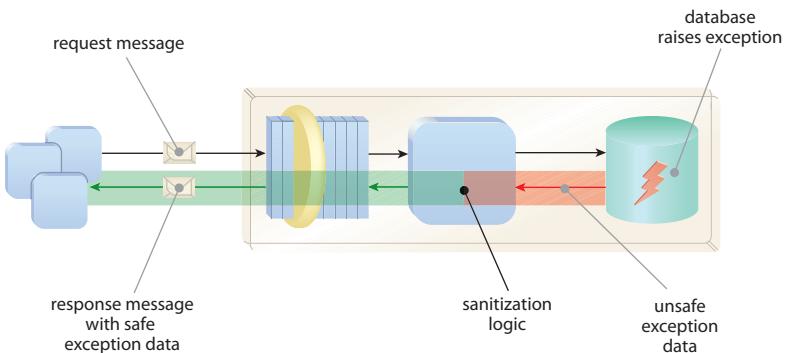


Figure 13.2

Potentially unsafe information is sanitized by routines added to the service logic, thereby releasing only safe exception information to service consumers.

The exception sanitization process can be further formalized, allowing output messages containing exception details to be standardized across services. This enables maintenance staff to troubleshoot and centrally manage exceptions on an inventory-wide basis.

Application

Sanitization routines can be applied at design-time during the initial service delivery or as part of a service refactoring effort. This results in pre-defined exception details that are considered “safe by design.” However, this pattern is also focused on the incorporation of run-time exception shielding logic.

The basic exception shielding process with runtime sanitization logic occurs as follows:

1. The consumer submits a request message to the service.
2. The service attempts to process the request and throws an exception. The exception may contain safe or unsafe information.
3. Exception shielding routines residing in the service logic check the exception information. If it is safe by design, it is already considered sanitized and is returned to the consumer unmodified. If the exception is identified as unsafe, it is replaced with safe exception information.
4. The service returns the safe exception message to the consumer.

Exception shielding routines are commonly built into individual services because it is generally preferable to keep shielding logic as close to the service as possible. For this purpose, the shielding routines may be part of the core service logic or separated into an intra-service agent or handler.

This pattern can also be applied in combination with Utility Abstraction (168) in order to centralize common shielding logic into a utility service that can be reused by other services. However, this approach would usually be used as a “safety net” to complement existing exception shielding logic within the service.

Note that an unhandled exception can be wrapped by another exception. The exception shielding logic needs to be sophisticated enough to ensure that all outer exceptions are checked for wrapped exceptions before allowing them to be returned to the consumer.

Unsanitized exception data can also be safely captured in an event log, allowing maintenance staff to identify and troubleshoot exceptions. This type of information can assist with intrusion detection and incident response, and monitoring tools can further capture and respond to safe exception information by automatically notifying administrators when

these exceptions occur. The event log itself must, of course, be secured to prevent unauthorized access.

This same information can also be used by developers to diagnose design-time errors. In some cases, tools might require that a given fault message contains an ID that help desk staff can use to more effectively troubleshoot and trace problems. The exception shielding logic can be designed to generate such an identifier for each exception.

Impacts

Because the exception information provided to consumers is sanitized, it lacks details that can be valuable when trying to track or trace error information. Often, consumers are given a GUID that must subsequently be searched for in error logs in order to retrieve exception details necessary to resolve certain exceptions. Being able to dynamically turn exception shielding functions on and off can help alleviate these situations.

Also while exception shielding logic can increase the amount of runtime filtering and processing the service must perform, because this processing is only instantiated when exceptions actually occur, the impact is trivial and should not affect regular service operation.

Furthermore, developers building exception shielding logic require an understanding of the range of potential security threats this logic is intended to protect the service from. Otherwise, incomplete or ineffective exception logic can lead to a false sense of security.

Relationships

Exception Shielding represents a form of utility logic that can be further supported by Service Agent (543), Utility Abstraction (168), and Service Perimeter Guard (394) when isolating it especially for the purpose of reuse across services.

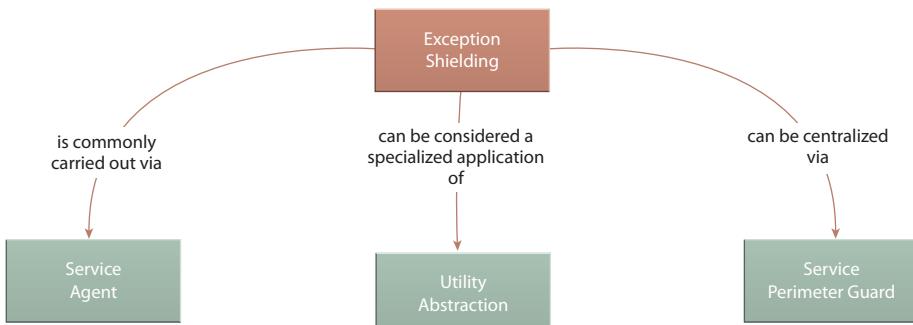


Figure 13.3

Exception Shielding shares the same types of relationships as other utility-centric patterns.

CASE STUDY EXAMPLE

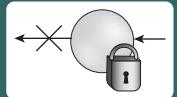
External hackers have been regularly triggering exceptions in the Retail Lumber service that accesses Alleywood's retail database. The database had been responding by providing detailed information about the nature of the exception and a number of environmental details about the surrounding implementation. The Retail Lumber service was unwittingly relaying these details to malicious consumers via SOAP fault messages.

As part of the redesign project, Alleywood architects incorporate runtime exception shielding routines to prevent unintentional information within unhandled exceptions from being exposed externally. Because Alleywood's operations team requires exception information in order to troubleshoot legitimate issues, these routines first write exception information to a secure central log where it can be analyzed. Each of these log entries includes a unique identifier that is included with the sanitized general error message that replaces each exception. When partner organizations contact the help desk, this number can be used to trace back to the specific exception that caused the issue.

Alleywood was fortunate because the application server upon which their Web services were running allowed for pre and post-message processing logic to be inserted declaratively into the SOAP message processing pipeline. As a result, Alleywood was able to standardize the exception shielding logic and then reconfigure each of their services to utilize this new behavior.

Message Screening

By Jason Hogg, Don Smith, Fred Chong, Tom Hollander, Wojtek Kozaczynski, Larry Brader, Nelly Delgado, Dwayne Taylor, Lonnie Wall, Paul Slater, Sajjad Nasir Imran, Pablo Cibraro, Ward Cunningham



How can a service be protected from malformed or malicious input?

Problem	An attacker can transmit messages with malicious or malformed content to a service, resulting in undesirable behavior.
Solution	The service is equipped or supplemented with special screening routines that assume that all input data is harmful until proven otherwise.
Application	When a service receives a message, it makes a number of checks to screen message content for harmful data.
Impacts	Extra runtime processing is required with each message exchange, and the screening logic requires additional, specialized routines to process binary message content, such as attachments. It may also not be possible to check for all possible forms of harmful content.
Principles	Standardized Service Contract
Architecture	Service

Table 13.2

Profile summary for the Message Screening pattern.

Problem

If a service-bound message contains invalid data, it can cause the service or other downstream systems that process the received data to behave in an undesirable manner (Figure 13.4). This data may be accidentally inserted by a consumer or intentionally added by an attacker.

NOTE

Sending harmful content to a program in this way is known as an *injection attack*.

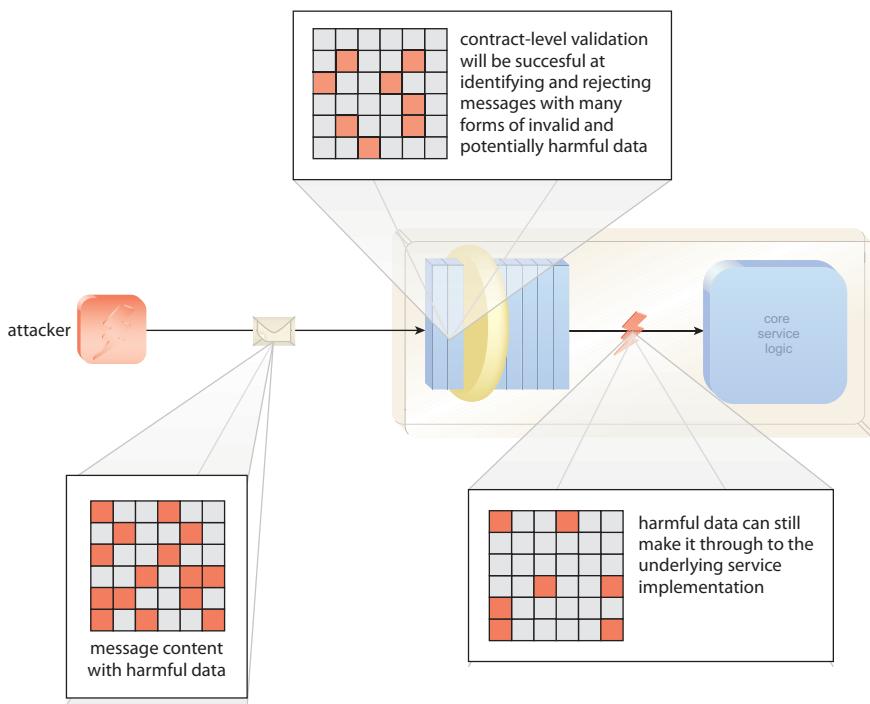


Figure 13.4

Parts of the message content transmitted by an attacker can make their way through to the service implementation. (The red squares represent potentially harmful data.)

Solution

When designing the service logic, it is assumed that all input data is harmful until proven otherwise. Specialized threat screening routines are therefore added to the internal service logic. The routines in this logic enforce well-defined policies that specify which parts of a message are required for the service to process the request. Because these filtering routines reside within the service, it remains protected without reliance on consumer-side validation logic (Figure 13.5).

Application

Applying this pattern requires that the necessary message screening routines be added to the service in such a manner that they are invoked when input data is received by any service capability. These routines will generally be designed to perform a set of standard screening tasks, such as:

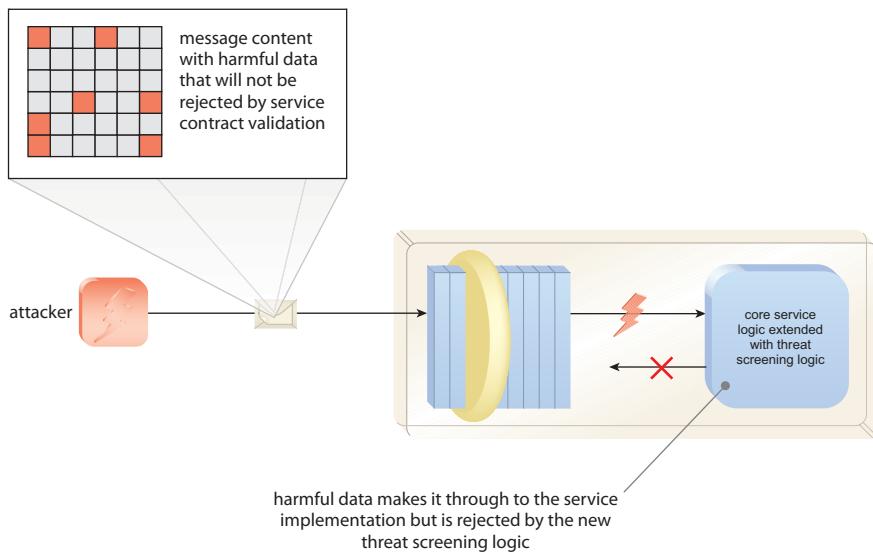


Figure 13.5

Because the service logic is equipped with extra message screening routines, malicious or malformed data can still be detected and rejected before it has a chance to do harm.

- Comparing the size of the request message against the maximum allowable size that is specified for request messages for a given capability.
- Parsing the entire request message for malicious content. (For Web services, malicious content could be placed in either the SOAP message header or body, so both would need to be checked.)

When designing threat screening logic, a number of considerations need to be taken into account, as follows:

- If a message is encrypted with message layer security, it may not be possible to inspect data for malicious content unless the message is decrypted beforehand or the screening logic has access to the decryption key.
- Custom threat screening logic is required to check binary message content, such as attachments. Such logic must be capable of recognizing each type of binary attachment that it encounters to ensure that it is free of malicious content. This form of binary data validation will often require the involvement of anti-virus filters or similar mechanisms.

- Message screening logic must be very efficient when it conducts its validation checks. Otherwise, it can turn into a system bottleneck and might itself become the target of a denial of service attack. A balance needs to be attained whereby the maximum message size is large enough to allow legitimate messages to be accepted, but small enough to prevent attacks.
- There is the option to combine this pattern with Utility Abstraction (168) or Service Agent (543) so as to isolate message screening logic into a separate utility service or an external intermediary service agent. However, this can establish a potential single point of failure that may become the target of aggressive attacks. It is often preferable to keep exception shielding logic close to the service and to consider this approach as a “safety net.”
- XML schemas can be further enhanced in support of this pattern by reducing the use of coarse-grained data types (such as `xsd:string`), which can be more prone to accepting a wider range of potentially harmful data. Similarly, XML message payloads that contain a CDATA field can be used to inject illegal characters that are ignored by the XML parser. If CDATA fields are necessary, they need to be inspected for malicious content.

So far we have been focusing on the receipt of input data via request messages issued by service consumers. It is also worth noting that harmful data could be obtained by the service implementation while it is responding to a legitimate consumer request message.

If the service logic, for example, receives data via another (less secure) service or by accessing non-secured data sources, it may inadvertently become a potential carrier of malicious input. In this case, it is the consumer receiving the response message from the service that is at risk. This scenario can be mitigated by applying this pattern to data received by the service implementation from non-trusted sources.

Impacts

Building and maintaining message screening logic requires specialized skills to ensure that as many threats as possible can be checked for. Depending on the extent to which screening logic is designed and how well the routines are actually built, this pattern may be only partially effective and could easily lead to a false sense of security. It is therefore important to design message screening routines in conjunction with a formal threat model.

Furthermore, the extra runtime processing required to thoroughly check incoming data for a range of security threats can be demanding and may introduce latency. This impact can

be somewhat reduced by deferring this logic to a highly autonomous and scalable utility service.

Depending on the nature of the surrounding infrastructure, it may simply not be possible to perform all desired checks. For example, service capabilities that accept binary attachments may not be able to validate every type of potential attachment format.

Relationships

Because it represents another form of utility-centric processing logic, Message Screening shares the same relationships as Exception Shielding (376).

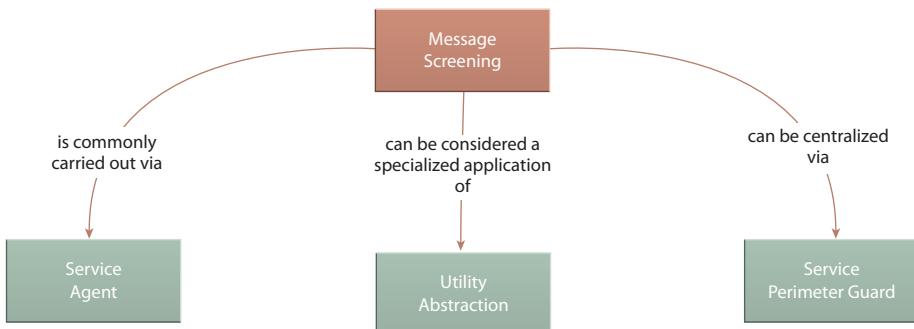


Figure 13.6

The potential pattern relationships of Message Screening are all associated with its separation and isolation because this form of logic can often be reused.

CASE STUDY EXAMPLE

In addition to the exceptions raised during the attacks explained in the previous case study example, Alleywood's Retail Lumber service had also been receiving a range of messages that contained both malformed and malicious content. Some of this data had been accidentally sent by a malfunctioning consumer; however, most was clearly part of a coordinated injection attack on the service. This part of the attack was attributed to the majority of problems that eventually occurred because so much harmful data was readily accepted as valid input.

A McPherson security specialist works with the development team in order to establish a set of screening routines that would be carried out with each incoming message received by the Retail Lumber service. Architects get involved, and soon a discussion

ensues about whether it would make more sense to abstract this logic into its own utility service that could be reused by the Retail Lumber service and others.

Alleywood architects feel this approach would make sense from an inventory standardization and normalization perspective and consider a number of different options including developing a new utility service that would implement the screening logic. Subsequent to some further modeling effort it is revealed that this design would require that consumers first call this service prior to the Retail Lumber service. The McPherson security consultant quickly points out that services cannot trust consumers in this manner.

As a result, the development team is asked to proceed with incorporating the screening logic directly into the Retail Lumber service design. Rather than having each service implement this functionality into its business logic, the architects create a single message validation routine and then declaratively specify that this logic use the same message processing interception mechanism within which exception shielding logic is applied—only this time the message screening logic is run before the message is processed.

Trusted Subsystem

By Jason Hogg, Don Smith, Fred Chong, Tom Hollander, Wojtek Kozaczynski, Larry Brader, Nelly Delgado, Dwayne Taylor, Lonnie Wall, Paul Slater, Sajjad Nasir Imran, Pablo Cibraro, Ward Cunningham



How can a consumer be prevented from circumventing a service and directly accessing its resources?

Problem	A consumer that accesses backend resources of a service directly can compromise the integrity of the resources and can further lead to undesirable forms of implementation coupling.
Solution	The service is designed to use its own credentials for authentication and authorization with backend resources on behalf of consumers.
Application	Depending on the nature of the underlying resources, various design options and security technologies can be applied.
Impacts	If this type of service is compromised by attackers or unauthorized consumers, it can be exploited to gain access to a wide range of downstream resources.
Principles	Service Loose Coupling
Architecture	Service

Table 13.3

Profile summary for the Trusted Subsystem pattern.

Problem

When underlying service resources, such as databases, can be accessed directly by consumer programs, the security of the resource can be compromised by malicious attackers and/or consumer programs can form unhealthy dependencies on parts of the service architecture that can lead to negative forms of consumer-to-service coupling (Figure 13.7).

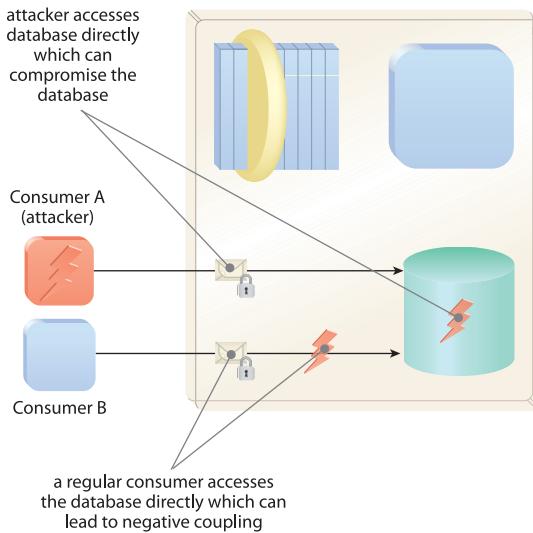


Figure 13.7

Allowing either malicious and non-malicious consumers to access the service's database directly leads to significant problems.

Solution

The service acts as a trusted subsystem of its underlying resources. Consumers can only access the resources via the service and the service uses its own credentials instead of the consumer's credentials to carry out access to the resources.

NOTE

This pattern also addresses the problem of when delegation is simply not supported by a service architecture (which in itself is a common situation).

Application

The service is positioned as the sole means by which the underlying resources can be accessed by service consumer programs. This will frequently require the joint application of Contract Centralization (409).

Consumers are further limited to authentication and authorization via the service and their credentials are not delegated to the underlying resource. Instead, the service uses its own credentials.

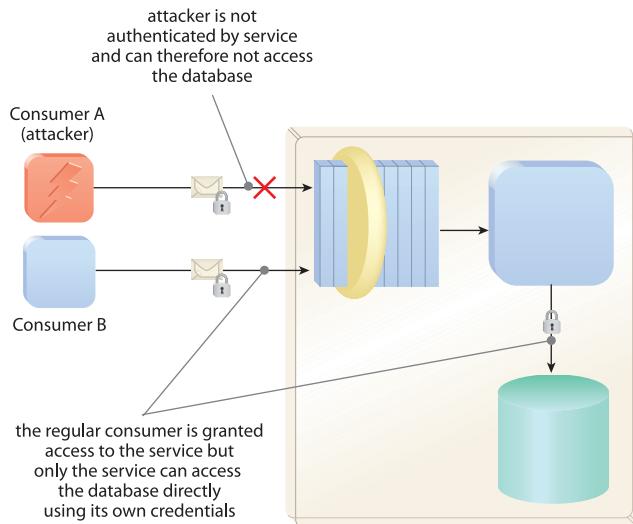


Figure 13.8

Neither a malicious or non-malicious consumer can access the database directly. Only the service itself can access the database with its own credentials.

When accessing a remote resource, the service must be designed to carry out the following steps upon the arrival of a request message with credentials:

1. Authenticate and authorize the message via Direct Authentication (656) or Brokered Authentication (661).
2. Send a request to the remote resource accompanied by the service's own credentials (or the service account under which the trusted subsystem process is being executed).
3. Upon receiving and processing a response from the resource, issue its own response back to the consumer.

To carry this out successfully, the remote resources must be able to verify that the mid-stream caller (the service) is trusted and not just any system process. Requiring this type of verification enhances security by making it more difficult for attackers to simulate a trusted subsystem and perform “man-in-the-middle” attacks.

As referenced earlier, each subsystem establishes a trust boundary. When multiple services are composed together to solve more complex problems, each can simultaneously act as a trusted subsystem and the resource that is accessed by a trusted subsystem. Figure 13.9 illustrates how this scenario establishes two overlapping trust boundaries.

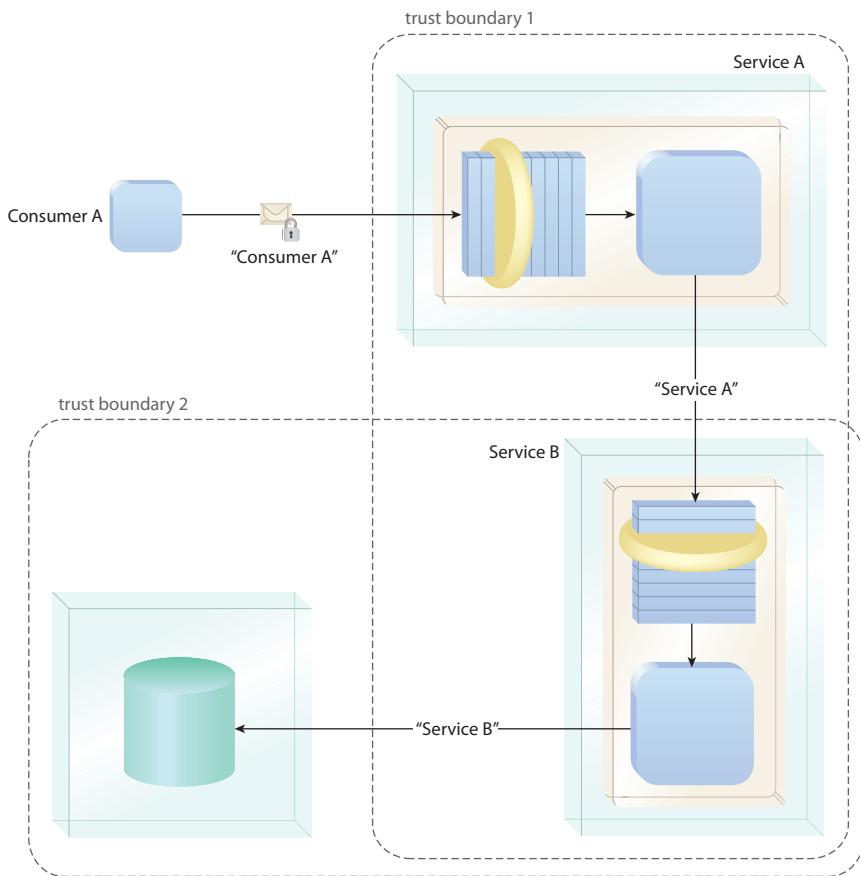


Figure 13.9

Service B acts as a trusted subsystem and also as the resource of a different trusted subsystem (Service A).

Several approaches and technologies can be used to implement this pattern:

- *Service accounts are used within the trusted subsystem* – A common method of implementing verification with the Kerberos protocol is to use a service account that is effective only within a particular trusted subsystem.
- *Local accounts are used on each host* – When it is not possible to authenticate using Kerberos service accounts, you can create a local account on each host within the trusted subsystem. These types of accounts are often referred to as “mirrored accounts,” as each will have the same login and password. Mirrored accounts generally require complex passwords that need to be frequently changed.

- *An X.509 PKI is used for authentication within the trusted subsystem* – The X.509 PKI can issue a certificate for each application within a trusted subsystem. To access resources the service must use an X.509 certificate as the basis for authentication. In addition, the certificate must be on the list of certificates that are authorized to access the resource.
- *IPSec is used between computers in the trusted subsystem so that communication is secure* – IPSec secures messages between two hosts at the network layer to provide data confidentiality, data integrity, and replay detection. It can be configured to initiate secure communication with the Kerberos protocol, X.509 certificates, or a pre-shared key. IPSec performs considerably better than message-layer security, but it does not allow for granular control of resources. This is because, with IPSec, a trusted subsystem can only be established between computers that participate in the trusted subsystem and not on a specific program accessing a specific resource.

NOTE

In some situations, resources might need to perform actions based on the identity of the consumer. For example, a database may require the consumer identity to enable data entitlement logic or to create an audit trail. In this case, the consumer's identity will still need to be "flowed" to the backend resource. Flowing the identity of a consumer while avoiding delegation can be performed by including a unique consumer identifier within either the message body or a custom SOAP header.

Impacts

If a service implementing Trusted Subsystem is compromised, it can be used to exploit any downstream resources it has access to. For this reason, services acting as trusted subsystems often become prime targets for attackers to probe for vulnerabilities within the enterprise.

Relationships

Because this pattern pertains to the authentication of consumers, it is naturally associated with Direct Authentication (656) and Brokered Authentication (661) in that when an alternative to delegation is required it is applied in combination with one of these two patterns.

Trusted Subsystem essentially acts as an extension to Contract Centralization (409) by reinforcing centralized service contract access with secure, centralized access to backend service resources.

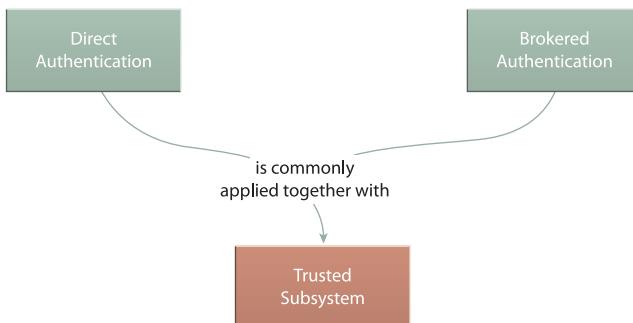


Figure 13.10

Even though Trusted Subsystem can be applied with other authentication-centric patterns, its role is to limit the propagation of the consumer credentials that are authenticated.

CASE STUDY EXAMPLE

The initial version of the Retail Lumber service exposed only basic catalog functionality, providing read-only information similar to what was already on the HTML-based Web site. As a result, Alleywood's architects did not feel that their services required authentication. But due to increased demand by business partners for access to more functionality, such as order processing and tracking, they must now consider additional security mechanisms including authentication.

The security team recommends that the service architecture be further enhanced, and in their report, they raise a series of issues:

- How will the external consumers be authenticated?
- Specifically which backend resources will consumers need direct access to?
- How should consumer identity information be propagated to backend systems?
- What auditing requirements exist?
- Where are dependent resources located within the corporate network, and what requirements exist for authenticating such resources?
- What (if any) advantages can be utilized by resource sharing techniques such as connection pooling?

The architecture team reviews each of these questions to better define the access and security requirements for the planned authenticated consumer message requests.

Subsequent to reviewing the results of their analysis together with the security team, it is determined to proceed with an authentication design whereby consumer credentials are not used for direct access to any service resources.

Instead, a service account is established. Calls from the Retail Lumber service to back-end databases will not access resources directly under the identity of the caller but will instead be transitioned to the service account. This approach also has the advantage of enabling connection pooling for use between the service and its databases.

However, Alleywood's architects also had to consider how to support data entitlement rules, meaning that the calls from the Retail Lumber services had to incorporate a unique identifier for the originating user, allowing the backend databases to only return information relevant to that consumer. Alleywood decides to pass this information in a custom SOAP header. McPherson's security specialists also point out that it is critical to ensure this custom SOAP header is also signed to guarantee that the identifier is not modified in transit. They therefore further proceed with Data Origin Authentication (649).

Service Perimeter Guard

By Jason Hogg, Don Smith, Fred Chong, Tom Hollander, Wojtek Kozaczynski, Larry Brader, Nelly Delgado, Dwayne Taylor, Lonnie Wall, Paul Slater, Sajjad Nasir Imran, Pablo Cibraro, Ward Cunningham



How can services that run in a private network be made available to external consumers without exposing internal resources?

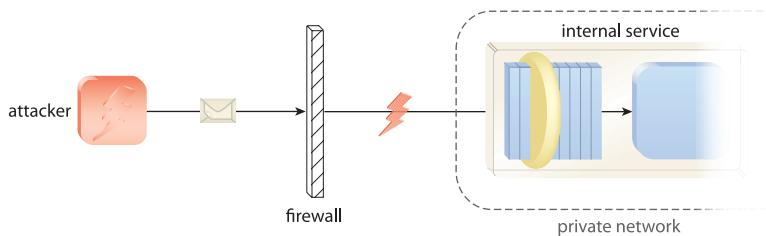
Problem	External consumers that require access to one or more services in a private network can attack the service or use it to gain access to internal resources.
Solution	An intermediate service is established at the perimeter of the private network as a secure contact point for any external consumers that need to interact with internal services.
Application	The service is deployed in a perimeter network and is designed to work with existing firewall technologies so as to establish a secure bridging mechanism between external and internal networks.
Impacts	A perimeter service adds complexity and performance overhead as it establishes an intermediary processing layer for all external-to-internal communication.
Principles	Service Loose Coupling, Service Abstraction
Architecture	Service

Table 13.4

Profile summary for the Service Perimeter Guard pattern.

Problem

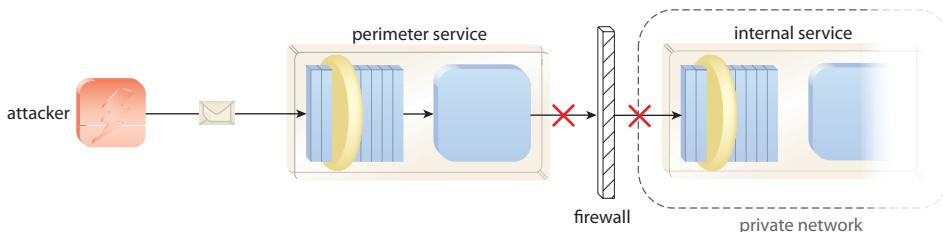
External consumers require access to one or more services deployed in a private network. Direct access to the private network would expose services to external attackers that can gain internal information and use it to compromise the services and the network (Figure 13.11).

**Figure 13.11**

A consumer that gains access to an internal service can exploit it directly through attacks.

Solution

An intermediary service is positioned at the perimeter of the private network and is established as the sole contact point for external consumers on behalf of one or more internal services (Figure 13.12).

**Figure 13.12**

The perimeter service processes the attacker's message and upon determining its malicious intent, rejects it. This spares the underlying internal service from exposure and unnecessary security-related processing.

Application

This type of service is typically deployed in a perimeter network (also known as the DMZ or demilitarized zone), which has access to resources in the private network through a firewall. It operates at the application layer and is intended to work in conjunction with existing firewall technologies (and not to replace them).

An external consumer will send a request message addressed to the perimeter service's external contract, which the perimeter service then forwards to the appropriate internal service. Similarly, when the internal service responds, the perimeter service relays the response to the external consumer. Throughout this exchange, the location and contract of the internal service remains hidden from the external consumer.

One of the primary advantages of Service Perimeter Guard is that it can establish centralized security processing on behalf of other services. This enables an architecture to be built around a perimeter service capable of implementing other security patterns, such as Brokered Authentication (661), Message Screening (381), and Exception Shielding (376).

NOTE

This pattern is similar in concept to Inventory Endpoint (260) but differs in two primary ways. First, Service Perimeter Guard is primarily about security-related processing, and its design solution is focused on providing a secure endpoint on behalf of other services. Secondly, Inventory Endpoint (260) is intended to specifically establish an entry point for an entire inventory of services. Service Perimeter Guard, on the other hand, has no such limitation. It can be used to represent one or several internal services. Note also that Inventory Endpoint (260) and Service Perimeter Guard can be applied to the same service.

Impacts

The use of perimeter services can add complexity and processing overhead and can further introduce performance bottlenecks when required to route and apply security processing to large numbers of messages.

As the single point of entry for a private network, this type of service can also become a primary target for attackers. This requires perimeter services to be thoroughly hardened. Also, the use of this pattern does not reduce the need to secure internal services, especially in relation to the communication that needs to occur between internal and perimeter services.

Relationships

The abstraction established by Service Perimeter Guard provides a point of isolation that can centralize Direct Authentication (656), Brokered Authentication (661), and Message Screening (381) on behalf of multiple other services. This pattern can further be combined with Inventory Endpoint (260) to establish centralized security processing for an entire inventory of services.

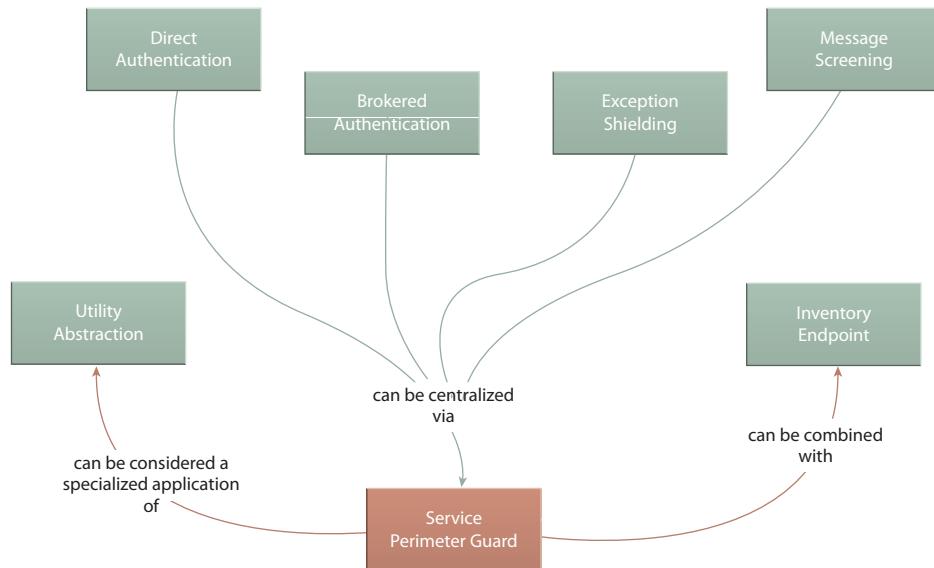


Figure 13.13

The security-centric wrapper service established by this pattern can centralize a variety of security-related processing logic.

CASE STUDY EXAMPLE

The redesign of the Retail Lumber service architecture is nearly completed when news of a new development project surfaces. Apparently, business requirements have emerged for Alleywood to also provide a Wholesale Lumber service to be made available for access to a new set of partner organizations in the lumber distribution sector.

Unlike the Retail Lumber service, which essentially provides an online access point for a broad public client-base, the Wholesale Lumber service will establish a programmatic interface for a limited number of partner organizations.

In order to avoid the problems encountered with the original Retail Lumber service, Alleywood decides to apply the same security patterns to the Wholesale Lumber service. However, after taking a look at the requirements specification for this service, it becomes apparent that some additional security measures need to be added.

Specifically, the service will also need to be accessed by other internal services that need to perform some of the same queries and orders for bulk lumber shipments to be issued

to various departments within Tri-Fold. Even though Tri-Fold has its own service inventory, it is technically not an external consumer, and this service could end up being part of several Tri-Fold service compositions.

The security team is uncomfortable with the idea of exposing a service to both external and internal consumers and recommends establishing a wrapper service that acts as the endpoint for external interaction only.

A perimeter service is subsequently designed to perform all authentication of external consumers. Architects further realize that the introduction of this service further provides an opportunity to further support the application of Exception Shielding (376), Trusted Subsystem (387), and Message Screening (381), by adding some extra “safety net” security processing logic.

This alleviates the Wholesale Lumber service from having to carry out most of this security-related logic and also prompts yet another redesign of the Retail Lumber service. The original idea architects had to move some of the more common security logic into a utility service is now justified. To accomplish this without impacting existing consumers, architects further apply this pattern together with Proxy Capability (497).

After the application of the four patterns described in this chapter, the Alleywood architects finally receive the green light to go live with the updated Lumber Order system.



Chapter 14

Service Contract Design Patterns

Decoupled Contract

Contract Centralization

Contract Denormalization

Concurrent Contracts

Validation Abstraction

Service-orientation places a great deal of emphasis on the design of service contracts. The Standardized Service Contract design principle in fact requires that all contracts within a given service inventory conform to the same conventions so as to establish a truly federated endpoint layer. From an inventory architecture perspective, this requirement is addressed by Canonical Schema (158) and Canonical Expression (275), as explained in previous chapters.

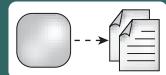
For example, both Decoupled Contract (401) and Contract Centralization (409) are considered essential to service design (especially when building services as Web services), because when combined, these patterns position the contract as an independent yet still central part of the service architecture.

The remaining patterns in this chapter provide various techniques for accommodating multiple consumer types. They can be applied independently or together. For example, Concurrent Contracts (421) can be used to establish multiple endpoints for a service, each of which can be further optimized via Contract Denormalization (414) and Validation Abstraction (429).

Furthermore, it is worth noting that Contract Denormalization (414) and Concurrent Contracts (421) can be part of the initial service design, or they can be applied after a service has been in use for some time. In the latter case, these two patterns could be considered governance-related as much as the design patterns in Chapter 16.

Decoupled Contract

How can a service express its capabilities independently of its implementation?



Problem	For a service to be positioned as an effective enterprise resource, it must be equipped with a technical contract that exists independently from its implementation yet still in alignment with other services.
Solution	The service contract is physically decoupled from its implementation.
Application	A service's technical interface is physically separated and subject to relevant service-orientation design principles.
Impacts	Service functionality is limited to the feature-set of the decoupled contract medium.
Principles	Standardized Service Contract, Service Loose Coupling
Architecture	Service

Table 14.1

Profile summary for the Decoupled Contract pattern.

Problem

Services can be built using component-centric distributed development technologies, such as .NET and Java. Although these development environments provide adequate platforms for building components as services, they usually require that the technical contract be physically bound to the underlying service logic when the service is built solely as a component. This essentially requires that the service contract be expressed via the same native technologies used to build the components (Figure 14.1).

As a result, the utilization and evolution of services is inhibited because they can only be used by consumer programs compatible with their technology. Even though bridging and transformation products are available, this limitation generally results in increased integration effort and a

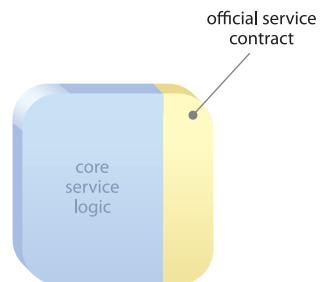


Figure 14.1

A component designed as a service, exposing an official service contract.

reduced number of potential service consumers (which translates into reduced reuse potential).

Furthermore, requiring that consumers be bound to a native implementation technology results in a negative form of coupling (known as technology coupling) that establishes direct dependencies on the continued existence of that technology. Should the service owner ever want to upgrade or replace the underlying logic with logic built using a different development platform, it would be very difficult to accomplish without effectively breaking all existing consumer dependencies (Figure 14.2).

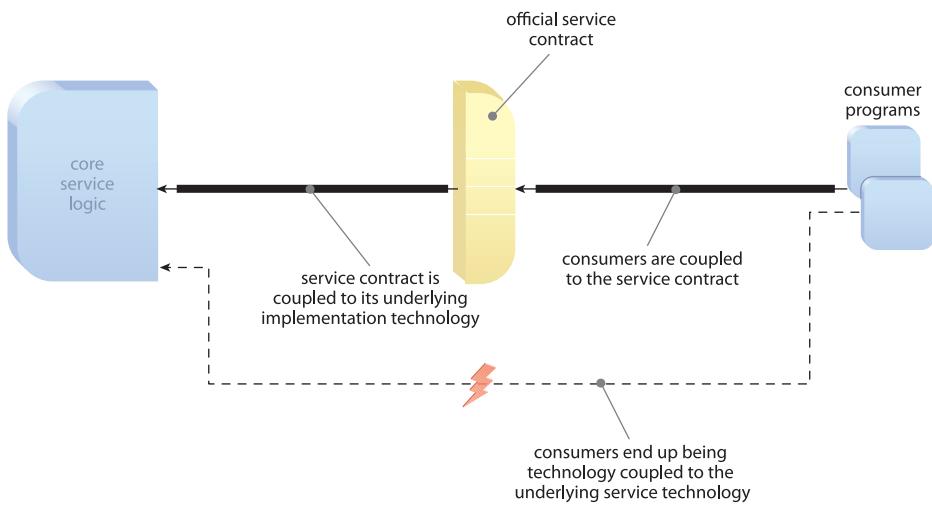


Figure 14.2

Because service consumers are required to couple themselves to a service contract that is itself coupled to the native component technology, the consumers become technology coupled.

Solution

The service contract is created as a physically separate part of the overall service implementation. This decouples the contract from the underlying service implementation, allowing it to be independently designed and governed (Figure 14.3).

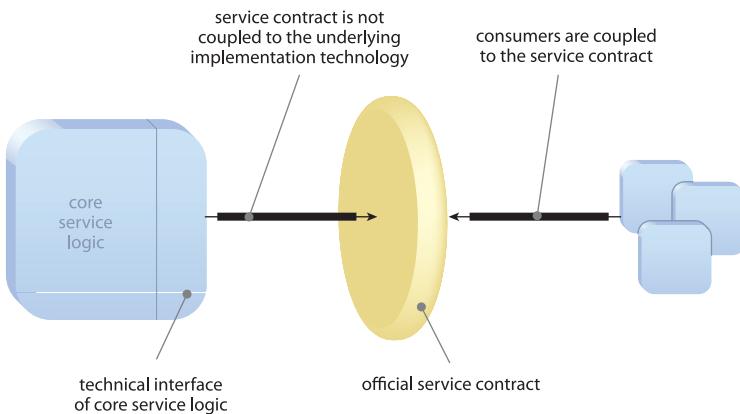


Figure 14.3

By decoupling the service contract, the service implementation can be evolved without directly impacting service consumers. This can increase the amount of refactoring opportunities and the range of potential consumer programs (and corresponding reuse).

Application

Web services represent the most popular means of applying this design pattern, as they force the service contract to be expressed in separate description documents using industry standard meta languages.

Service inventories based on the use of contracts that support industry standards are considered to have the greatest freedom for long-term governance and vendor diversification. Therefore, Web service contracts are effective as long as the underlying runtime platforms are deemed sufficiently mature to support the range of processing logic required by all service capabilities and any potential composition configurations in which they may need to take part (Figure 14.4).

A common risk associated with expressing service contracts using Web service technologies is the established approach of auto-generating Web service contract description documents (WSDL, XML Schema, and WS-Policy definitions) via modern development tools. This technique can result in implementation coupling, a negative coupling type whereby contracts express implementation details, such as physical data models or proprietary component method parameters.

Consumer programs that then bind to these types of service contracts form design-time dependencies on their physical implementation details. When the underlying service implementation is required to change, all existing service consumers can be immediately affected. This inflexibility can paralyze the evolution of a service and introduce the requirement for multiple, premature service versions (Figure 14.5).

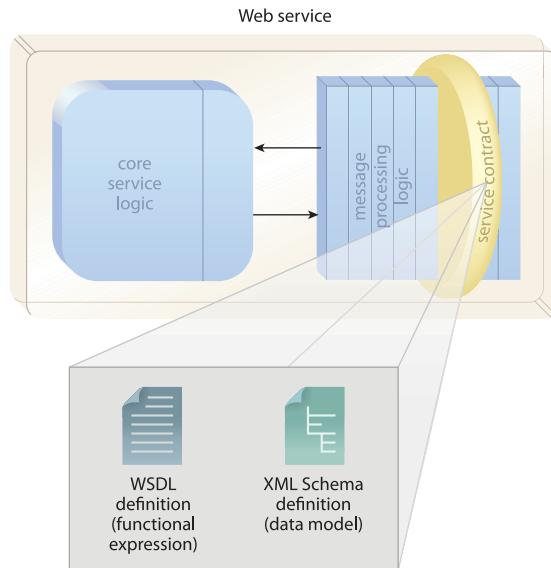


Figure 14.4

When the decoupling of a contract from its implementation is realized by delivering the service as a Web service, it introduces the need to formally define the functional expression and data representation parts of the service contract via WSDL and XML Schema (and optional WS-Policy) definitions.

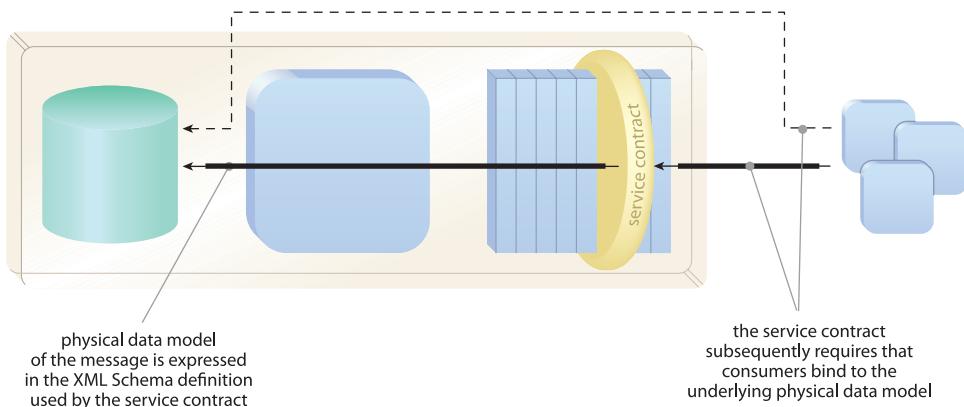


Figure 14.5

An example of how implementation coupling of the service contract can propagate implementation coupling to service consumers.

The Service Loose Coupling design principle addresses this issue by advocating the independent creation of a service contract so that the contract's content can also remain decoupled from any existing or future logic and resources it may be required to encapsulate. This further allows the contract to be shaped according to existing design standards (as per the Standardized Service Contract principle) and establishes it as an endpoint into service logic freed from ties to underlying implementation details (Figure 14.6).

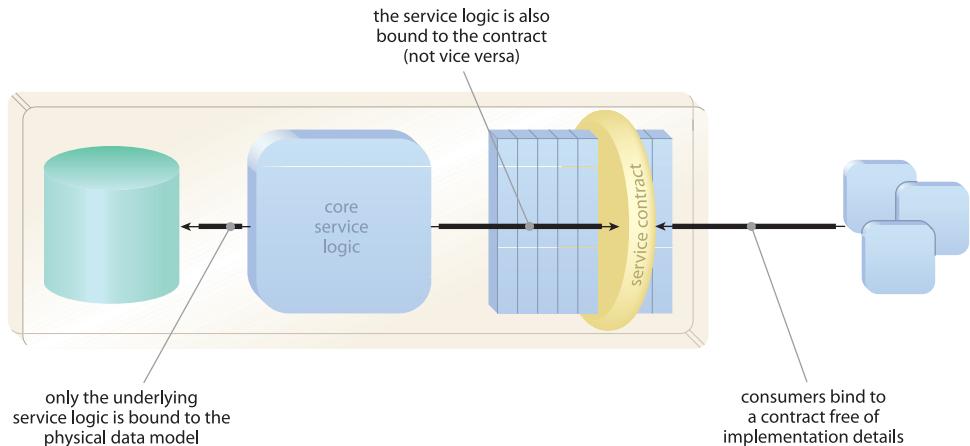


Figure 14.6

Revisiting the previous example, if the service contract can be kept free of implementation details, consumers can avoid binding to them.

Impacts

When decoupled contracts are employed, the service and its consumers will be tied to any limitations associated with the maturity of the vendor platform's support for relevant contract and related communications technologies. Deficiencies within the contract technology platform can inhibit the utilization of the service. Standardizing on a technology decoupled contract design can then impose any deficiencies on the service inventory as a whole.

Relationships

Decoupled Contract is fundamental to many design techniques that revolve around the use of Web services or directly benefit from the existence of a physically separate service contract.

Patterns associated with the post-implementation augmentation of service contracts, such as Service Decomposition (489), Proxy Capability (497), Distributed Capability (510),

and Contract Denormalization (414), all can be more effectively applied to a service with a decoupled contract.

Contract Centralization (409) and Service Refactoring (484) benefit tremendously from this pattern due to the independence it achieves between contract and implementation. Service Façade (333) is often applied to add another level of abstraction between core service logic and the decoupled contract, and these two patterns furthermore enable the realization of Concurrent Contracts (421).

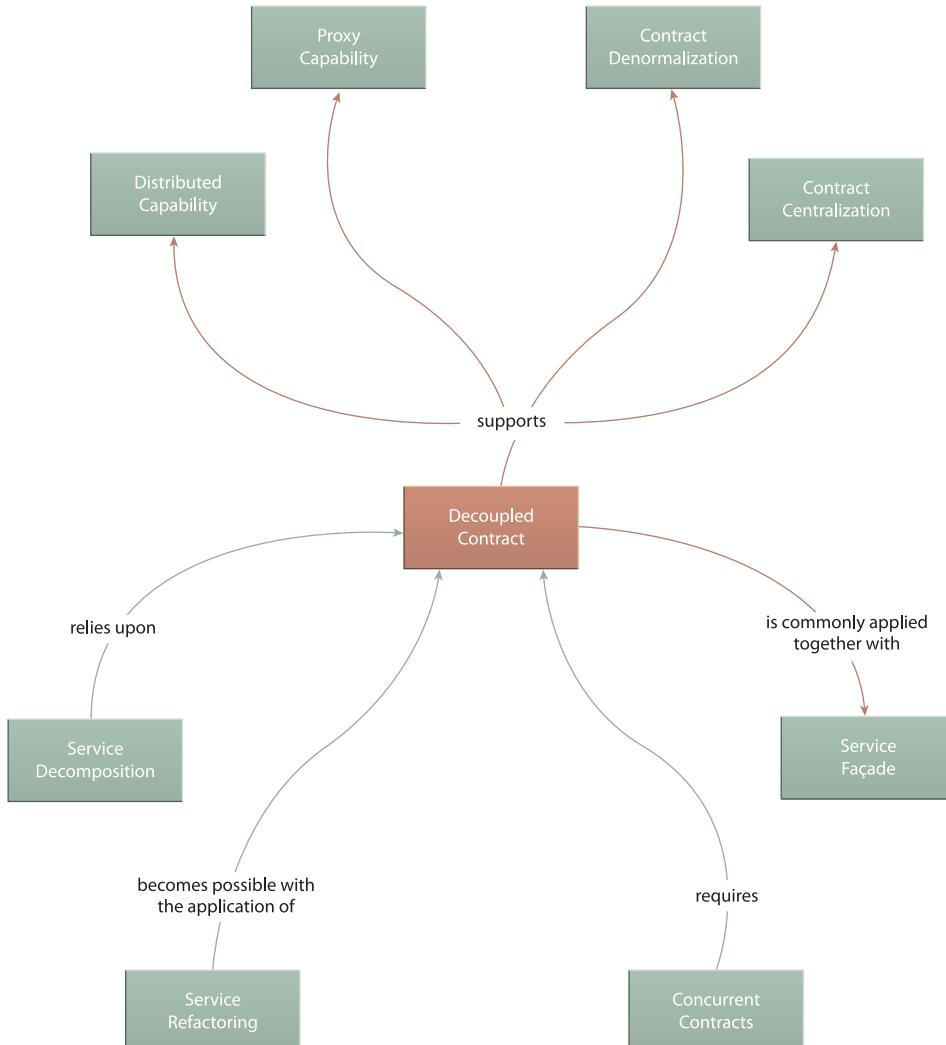


Figure 14.7

Decoupled Contract establishes an important separation of logic, technology, and technical interface that can be leveraged by many other design patterns.

CASE STUDY EXAMPLE

The case study example for Service Façade (333) in Chapter 12 introduced a utility component called “Data Controller” that is responsible for providing centralized access to a set of databases. Although the nature of this component is similar in concept to a utility service, the fact that it was delivered prior to the SOA initiative and does not conform to the design standards applied to newly developed services has prompted FRC architects to categorize it as a part of their legacy environment.

As explained in the Service Façade (333) example, this component was used by the Appealed Assessments service to provide the data it required to generate specific reports. As new projects emerge, new services are designed. One project in particular is tasked with delivering the Fines and Evaluations services. Upon a review of their processing requirements, it is determined that both of these services will require access to the databases represented by the Data Controller component.

The project team is reluctant to have each service couple itself to a legacy program that was recently changed (see the Legacy Wrapper (441) case study example in Chapter 12) and may still be subject to further change. Each of these modifications would impact each of the services, increasing the potential governance burden.

After some discussions with the owners of the Data Controller component, it is decided that this component will be redesigned as a utility *service* for inclusion in the FRC service inventory. One of the first challenges to address is the fact that the Data Controller exists as a standalone Java EJB. By way of Canonical Protocol (150), the FRC service inventory was standardized on the Web services technology framework, allowing all service endpoints to be comprised of decoupled service contracts (WSDL and XML Schema definitions).

For the Data Controller to comply to this requirement, it too needs to be equipped with a decoupled service contract that exists independently from its underlying service logic. Subsequent to a redesign, the Data Controller is deployed as a Web service. Its decoupled contract makes it possible for it to expose a standardized service contract independently from its implementation (Figure 14.8). This will allow the underlying logic and technology to undergo changes and refactoring efforts without affecting the contract and all of the service consumers that bind to it.

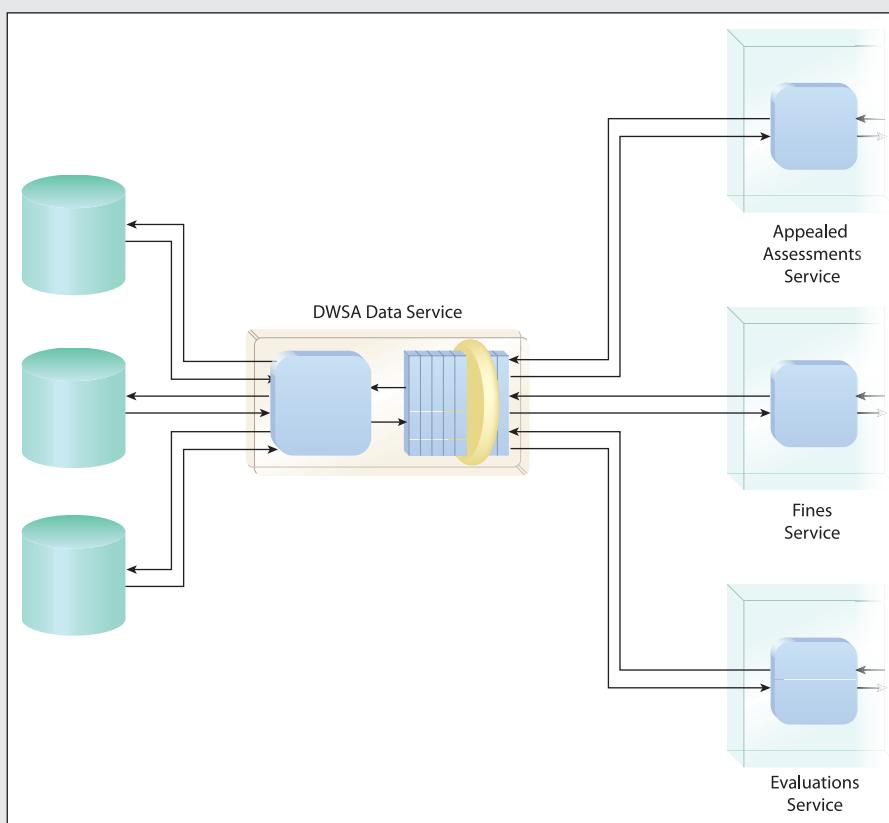


Figure 14.8

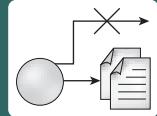
Appealed Assessments, Fines, and Evaluations services now all access the new DWSA Data service (formerly the Data Controller component) via its decoupled contract.

NOTE

As shown in Figure 14.8, the redesign also results in a new name for the service. Because of Canonical Expression (275), the service is given the name “DWSA Data,” which the architects feel better expresses its functional context. DWSA is short for “Data Warehouse Subdivision A,” the part of the overall data warehouse that provides a collection of statistical data relevant to appeals, fines, and assessments.

Contract Centralization

How can direct consumer-to-implementation coupling be avoided?



Problem	Consumer programs can be designed to access underlying service resources using different entry points, resulting in different forms of implementation dependencies that inhibit the service from evolving in response to change.
Solution	Access to service logic is limited to the service contract, forcing consumers to avoid implementation coupling.
Application	This pattern is realized through formal enterprise design standards and the targeted application of the Service Abstraction design principle.
Impacts	Forcing consumer programs to access service capabilities and resources via a central contract can impose performance overhead and requires on-going standardization effort.
Principles	Standardized Service Contract, Service Loose Coupling, Service Abstraction
Architecture	Composition, Service

Table 14.2

Profile summary for the Contract Centralization pattern.

Problem

Even when services within an enterprise are deployed with published, standardized service contracts, those designing consumer programs can be tempted to look for alternative entry points into service logic. For example, it may be easier or more efficient to bypass the service contract and simply access its underlying logic directly using native protocols (Figure 14.9).

Subsequently, the service contract loses its significance, and the service ends up with numerous tight dependencies (usually in the form of integration channels) to various parts of its implementation. This inhibits the evolution and governance of the service and undermines many of the objectives of service-orientation.

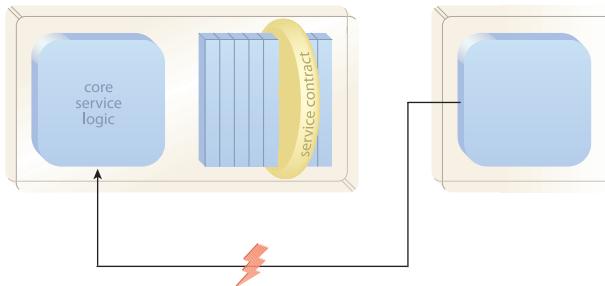


Figure 14.9

A service consumer program simply bypasses the service contract to access underlying logic directly.

Solution

Contract Centralization establishes a design standard that positions the service contract as the sole entry point into service logic. This allows for a consistent form of loose coupling with all service consumer programs (Figure 14.10).

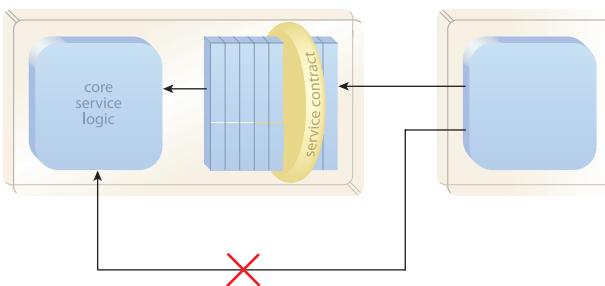


Figure 14.10

Through Contract Centralization we place the service contract front and center within a service architecture. This is why much of service-orientation is focused on contract design.

Application

The application of this pattern establishes a distinction between the official published service contract and other parts of the service that can also be accessed via separate technical endpoints. For example, a consumer could interface with an underlying service component via that component's API. This API still represents a technical contract, but not the "official" service entry point. This would therefore be considered a non-centralized form of service coupling.

If Contract Centralization is enforced to a meaningful extent, the service contract becomes a focal point for a great deal of interaction. From a long-term evolutionary perspective, therefore, Web services and REST services provide an effective means of establishing a centralized contract while remaining decoupled from the service's underlying environment, as per Decoupled Contract (401).

On the other hand, contract technology that requires the use of proprietary communication protocols will limit service access to those consumer programs compatible with the proprietary requirements of the contract. In this case, the repeated application of this pattern can result in a proliferation of technology coupling throughout an inventory.

Impacts

Integration architecture has a well-rooted history that predates the emergence of middleware and the EAI movement. Few of the past integration architectures were based on a concept like centralization, and more often than not, connections were made to whatever application entry points were the most convenient and efficient to fulfill immediate requirements.

Requiring architects, developers, project managers, and other team members to now forsake all of the options they've had in the past in favor of a design standard that is being established for the long-term good of the enterprise can be difficult. Resistance to centralization is common, and tactical requirements, such as time-to-market priorities and budget restrictions, can motivate some project teams to simply disregard this pattern altogether.

Furthermore, requiring that all service consumers access a body of logic through a single entry point can result in a classic convergence of performance issues, especially when having to reroute multiple existing integration channels to interface with a service contract. Contract Centralization needs to be expected and planned for, especially with agnostic services because they are subject to the greatest concurrency demands.

Relationships

By looking at the variety of relationships in Figure 14.11, it is evident how important Contract Centralization is to service-orientation. It is a part of establishing an effective endpoint layer within inventories and the repeated utilization of agnostic services, such as those based on Entity Abstraction (175) and Utility Abstraction (168), relies on the base requirement that they only be accessed via their contracts which, in turn, fully supports the long-term, independent governance of services subject to Service Refactoring (484).

Contract Centralization is responsible for positioning service contracts as a fundamental service access tier that can be further extended via complementary policy and schema layers, as per Policy Centralization (207) and Schema Centralization (200).

It is important to acknowledge that the centralization of service contracts is supported (and often enabled) by Decoupled Contract (401) and Service Normalization (131). Decoupled contracts can be much more easily centralized and separately positioned from underlying service implementations and the normalization of services further ensures that centralized contracts do not end up representing redundant logic.

One of the closest relationships is between Contract Centralization and Logic Centralization (136), as explored further in the description for the compound pattern Official Endpoint (711).

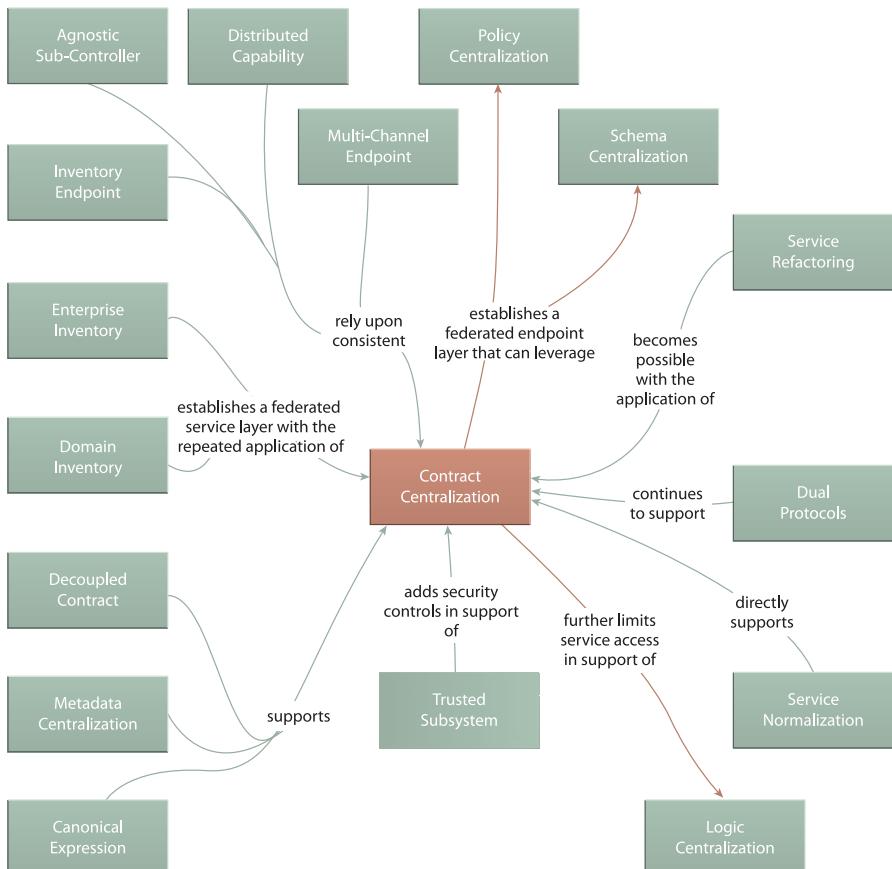


Figure 14.11

Contract Centralization is a lot like an enterprise design standard in that it regulates how services are accessed and therefore has relationships with both service logic and contract-related patterns.

CASE STUDY EXAMPLE

Enterprise architects within the FRC make a strategic decision to position the delivery of all new service contracts as the sole allowable entry points into the corresponding service logic. This requires a great deal of communication and some education for existing IT staff that have become accustomed to achieving interoperability via specialized integration channels.

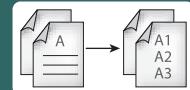
This move results in a formal, enterprise-wide design standard that affects all new services being delivered. Any legacy system logic a service capability may encapsulate can no longer be directly accessed by external applications. Although they expect resistance to this standard for some time, every effort is made to uphold it.

A remaining challenge, however, still looms. The legacy environments encapsulated by some of the planned services already have established integration channels with other applications. The team does not want to make the introduction of Contract Centralization too disruptive for the rest of the enterprise. At the same time, it wants to take any possible steps toward maximizing its independence to govern these services.

A decision is therefore made to assess each of the existing integration channels and to identify encapsulated systems with the most channels and the most integration-related activity. Those at the top of the list are scheduled for a transition toward supporting the contract-centralized architecture. The initial plan is to move them over within six to eight months, depending on available resources and other priorities.

Contract Denormalization

How can a service contract facilitate consumer programs with differing data exchange requirements?



Problem	Services with strictly normalized contracts can impose unnecessary functional and performance demands on some consumer programs.
Solution	Service contracts can include a measured extent of denormalization, allowing multiple capabilities to redundantly express core functions in different ways for different types of consumer programs.
Application	The service contract is carefully extended with additional capabilities that provide functional variations of a primary capability.
Impacts	Overuse of this pattern on the same contract can dramatically increase its size, making it difficult to interpret and unwieldy to govern.
Principles	Standardized Service Contract, Service Loose Coupling
Architecture	Service

Table 14.3

Profile summary for the Contract Denormalization pattern.

Problem

Because services can be utilized within a variety of compositions, it is difficult to express each capability in such a way that it is suited for each possible consumer program.

For example, a capability may not return a sufficient amount of data in response to a consumer request, or more commonly, it provides too much data, thereby imposing transmission and processing overhead upon the consumer program (Figure 14.12).

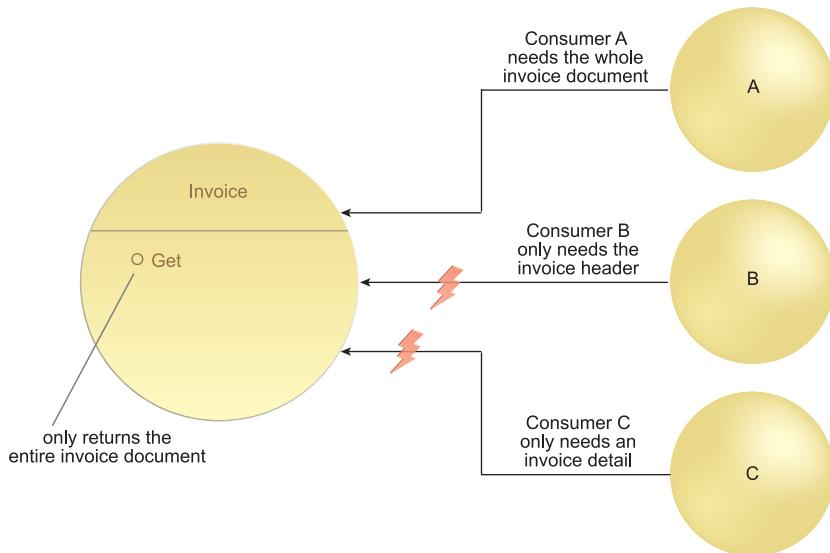


Figure 14.12

The Invoice service provides a Get capability that is not able to facilitate the varying granularity levels different service consumer programs prefer.

Solution

Unlike pursuing normalization across a service inventory, as per Service Normalization (131), where denormalization can impact the autonomy and governance of individual services, the level of acceptable normalization across a service contract is more flexible.

This flexibility allows for increased contract design options, including strategic incorporation of the denormalization of expressed functionality. In other words, the processing of one service capability does not need to be limited to one capability. Capabilities with redundant functionality offered at different levels of granularity can be provided to support multiple consumer and composition requirements (Figure 14.13).

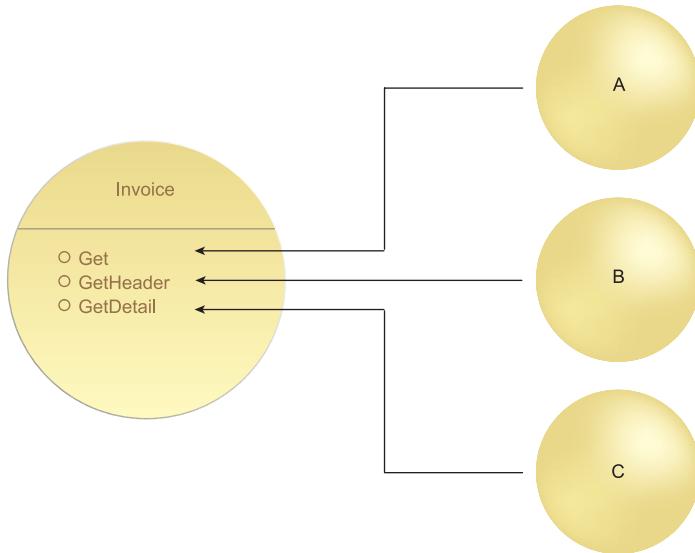


Figure 14.13

Equipped with additional (albeit redundant) capabilities, the Invoice service is able to better accommodate the individual requirements of the three consumers.

Application

Depending on the nature of the capability logic, there are some common ways that this pattern is applied:

- The same capability can be offered at different levels of granularity. As already illustrated in Figure 14.13, an entity service can contain different Get-related capabilities that can get an entire document, get just the document header, or get one or more document detail items (or a specific document property). The latter two variations introduce functionality that overlaps with the first and are therefore considered redundant.
- A new capability is added to an existing task service. Even though the task service encapsulates a body of business process logic, one or more capabilities can be added to expose segments of the process logic (normally in the form of modest subprocesses). This can establish alternate entry points into business process logic, but from an endpoint perspective, the capabilities appear to encapsulate redundant logic.

Even though this pattern intentionally introduces functional redundancy into the contract, there is typically no need to add significant functional processing to the underlying service logic. If all related capability definitions can be processed by the same set of components, the corresponding routines can be parameterized and shared.

Impacts

Overuse of this pattern can lead to overly large and convoluted service contracts. If multiple variations of each primary capability are added, the contract can become unmanageable and difficult to evolve. The effectiveness of agnostic services especially can suffer from poor functional expression.

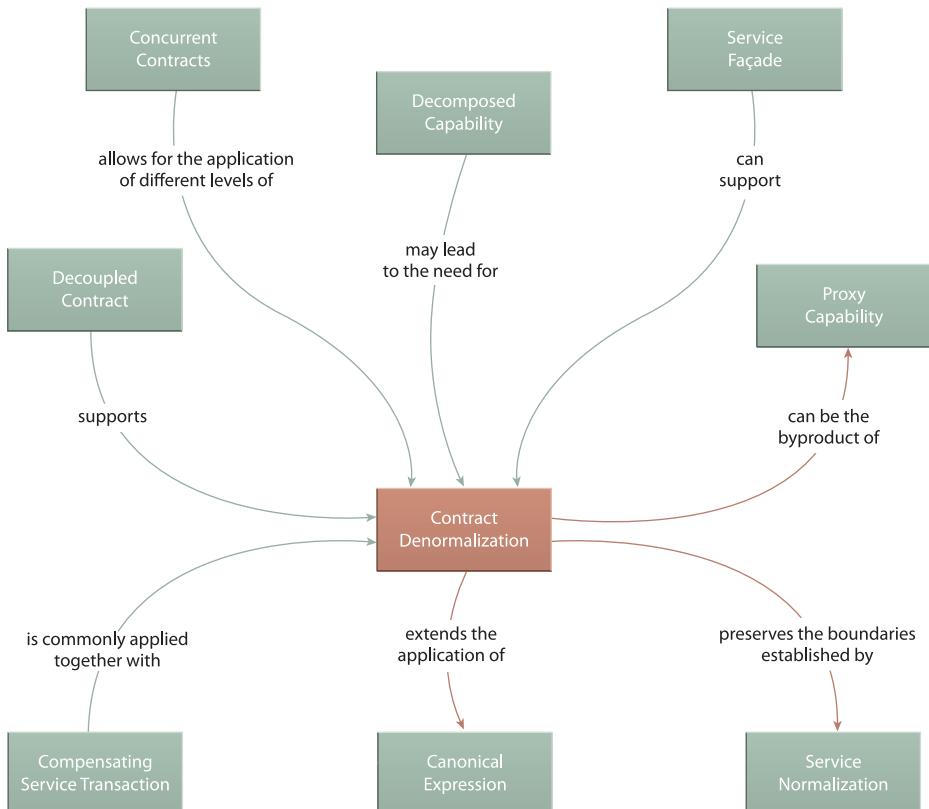
Furthermore, adding capability variations that expose redundant functionality may require the creation of multiple, also redundant schema definitions. A Web service implementation, for example, could easily be comprised of numerous schema files, making its governance increasingly challenging.

Relationships

Contract Denormalization introduces additional capabilities into a contract, most of which will repeat the expression of functionality. This is why Service Façade (333) is commonly applied in support of this pattern; it allows for a single façade component to interact with other components and routines on the service back-end in order to facilitate redundant contract capabilities without the need for redundant service logic.

The flexibility to apply this pattern is further increased via Decoupled Contract (401) primarily because it provides the freedom to fully customize a service contract independently for its underlying implementation.

Service Normalization (131) does not directly relate to the application of Contract Denormalization, but it is interesting to note that despite its name, this pattern does not interfere with the pursuit of normalizing service boundaries. The boundaries remain the same; only the contract content changes.

**Figure 14.14**

Contract Denormalization allows for the extension of contracts with redundant capabilities and therefore relates mostly to patterns that can support this requirement.

CASE STUDY EXAMPLE

The Officer entity Web service has been used by the FRC to provide centralized access to all processing and data associated with regulations officers. This service was initially delivered with a single Update operation that accepted a single complex type comprised of an entire officer record, including historical data, as shown here:

```
<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/"
            schemaLocations="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>
```

Example 14.1

A subset of the original Officer WSDL definition comprised of a single Update operation.

Limiting the service to this one operation was originally considered reasonable because the backend database used to house officer data was modeled in such a manner that it could only accept changes to historical log entries when those changes were accompanied by a number of parent values associated with the officer profile record itself.

However, the database in question was just replaced with a new product designed with a more flexible data model. In response to recent bandwidth concerns, a number of service contracts are revisited to explore the design of leaner, more optimized data exchanges.

When architects review the Officer Web service contract, they see an opportunity to provide a more streamlined input message format for when only updates to historical logs are required. Due to the change in data model, these logs can now be updated with only the parent officer ID value.

The development team then proceeds to denormalize the contract by adding a new operation named UpdateLog. As shown in Example 14.2, the UpdateLog operation is now part of the Web service interface, alongside the original Update operation.

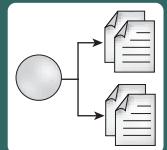
```
<definitions name="Officer" ...>
  ...
  <message name="UpdateOfficerLog">
    <part name="RequestB" element="off:OfficerLog"/>
  </message>
  <message name="UpdateOfficerLogConfirm">
    <part name="ResponseB" element="off:ReturnCodeB"/>
  </message>
  <portType name="OffInt">
    <operation name="Update">
      <input message="tns:UpdateOfficer"/>
      <output message="tns:UpdateOfficerConfirm"/>
    </operation>
    <operation name="UpdateLog">
      <input message="tns:UpdateOfficerLog"/>
      <output message="tns:UpdateOfficerLogConfirm"/>
    </operation>
  </portType>
  ...
</definitions>
```

Example 14.2

The revised, denormalized Officer WSDL definition with Update and UpdateLog operations.

Concurrent Contracts

How can a service facilitate multi-consumer coupling requirements and abstraction concerns at the same time?



Problem	A service's contract may not be suitable for or applicable to all potential service consumers.
Solution	Multiple contracts can be created for a single service, each targeted at a specific type of consumer.
Application	This pattern is ideally applied together with Service Façade (333) to support new contracts as required.
Impacts	Each new contract can effectively add a new service endpoint to an inventory, thereby increasing corresponding governance effort.
Principles	Standardized Service Contract, Service Loose Coupling, Service Reusability
Architecture	Service

Table 14.4

Profile summary for the Concurrent Contracts pattern.

Problem

By default, a service has a contract that expresses the full range of its abilities. However, it can be challenging to design this contract in such a manner that it accommodates different types of service consumers.

For example, the service contract may need to incorporate special processing extensions (such as policy assertions) not supported by all consumer programs. Or the service may need to be made available to semi- or non-trusted consumers that could potentially abuse some of its capabilities.

Having one contract support a range of consumer types is challenging both from design and governance perspectives and can ultimately lead to security concerns and constraints that limit the service's overall effectiveness (Figure 14.15).

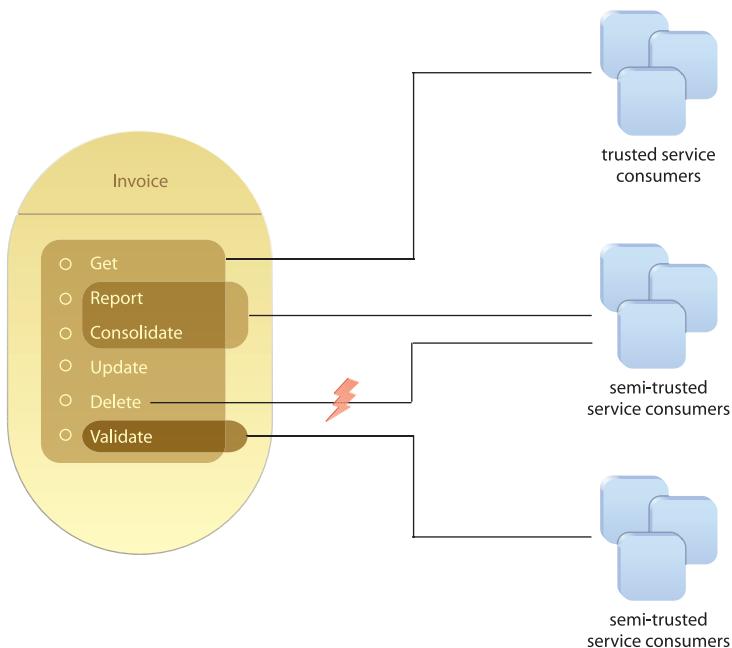


Figure 14.15

It can be undesirable to expose a service contract to all potential consumers, especially when some are less trusted than others. For example, exposing the full Invoice service contract to consumers that should only have access to a subset of its capabilities introduces the risk that consumers will attempt to access other capabilities anyway.

Solution

To accommodate different types of consumers, separate service contracts can be created for the same underlying service implementation. Even though this introduces redundancy in functional representation, it allows each contract to be extended and governed individually. It also provides the option of exposing a subset of the service capabilities to specific consumers (Figure 14.16).

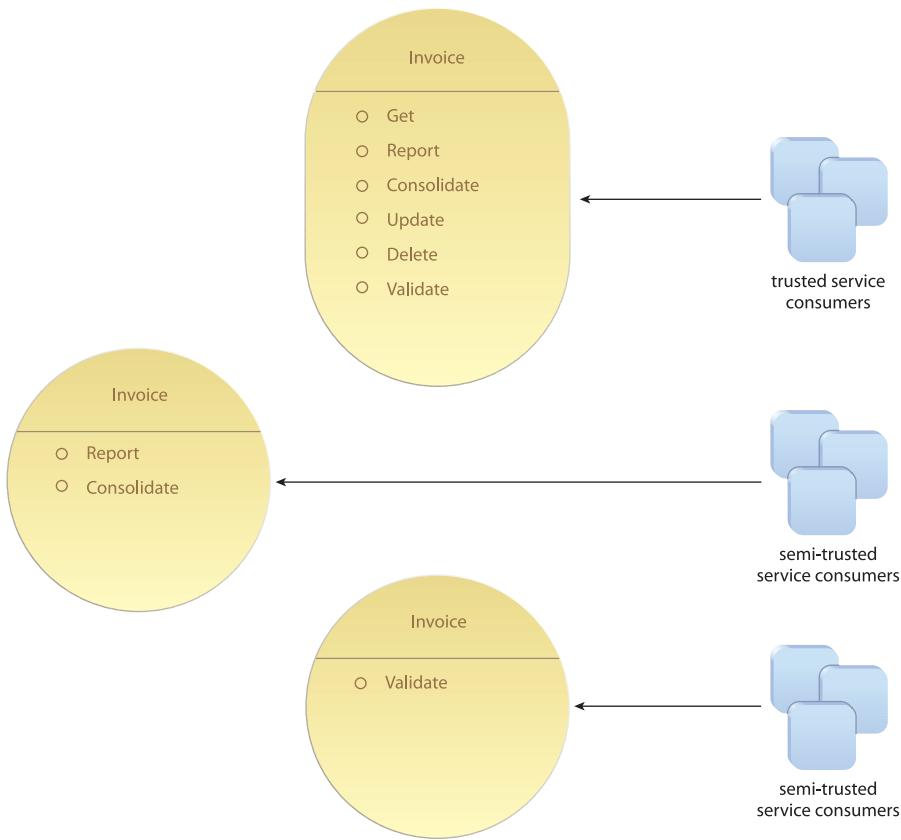


Figure 14.16

Separate contracts are created for the three service consumer categories. In some environments this may require the further qualification of the service name; for example, the three contracts may be named Invoice Admin, Invoice Reporting, Invoice Vendor.

Application

This pattern needs to be applied with care and moderation. Introducing multiple contracts leads to increased governance complexity and effort, as explained shortly in the *Impacts* section.

Often additional contracts are not considered necessary until well after the deployment of the service and its original contract. Therefore, it is recommended that new contracts not be created too reactively. Instead, each new consumer base (a group of related types of service consumer programs) should be well-defined so that it is confirmed that a new contract is warranted and that the nature of its expressed capabilities is clearly thought out.

The application of this pattern in general is more easily carried out if the service was originally designed according to Service Façade (333). A separate façade can actually be created for each new contract, as shown in Figure 14.17.

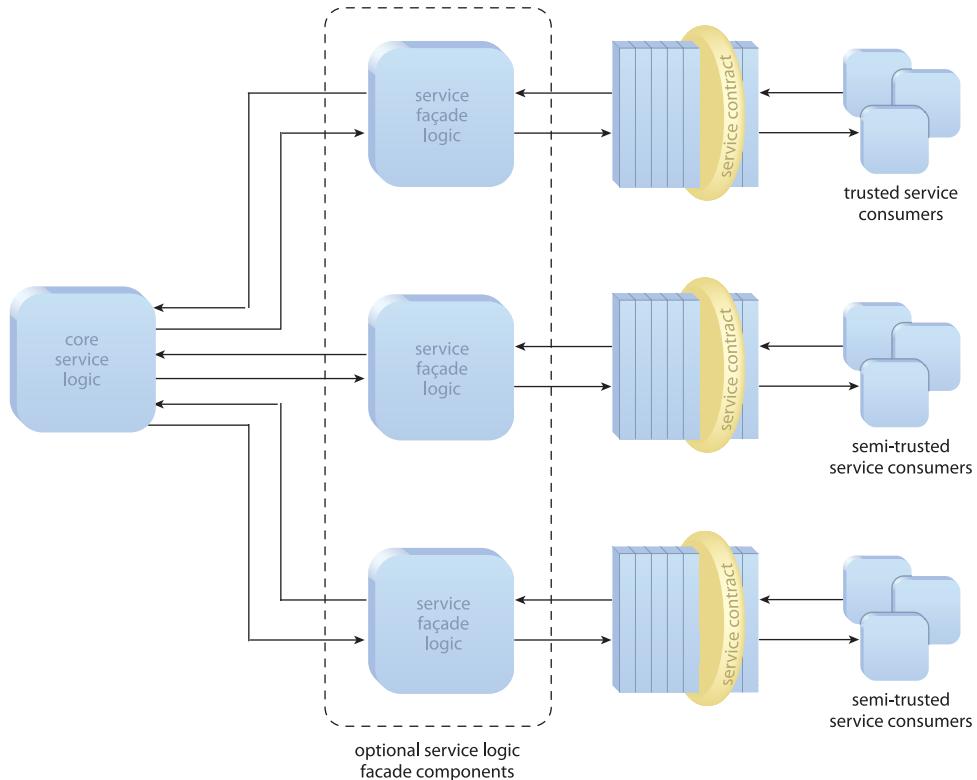


Figure 14.17

Multiple service contracts can be implemented with the support of Service Façade (333), which allows for contract-specific abstraction from the core underlying logic. This is especially relevant when contracts need to vary in terms of data representation support and security requirements.

There will be a natural tendency to want to modify the validation of contracts for different consumers. This also needs to be carefully assessed. Ideally, the contracts remain in alignment in how they express their respective capability sets. However, certain policies and security constraints may be applicable only to certain types of consumers (especially when having to facilitate both internal and external consumer bases).

Therefore, as a rule of thumb, it is best to limit the variation in validation logic to access control only. Validation logic based on underlying service business logic should remain the same across all contracts for the single service implementation. If different business rules or business constraints apply to different types of consumers, that decision logic may be best embedded within the underlying service logic, as per Validation Abstraction (429).

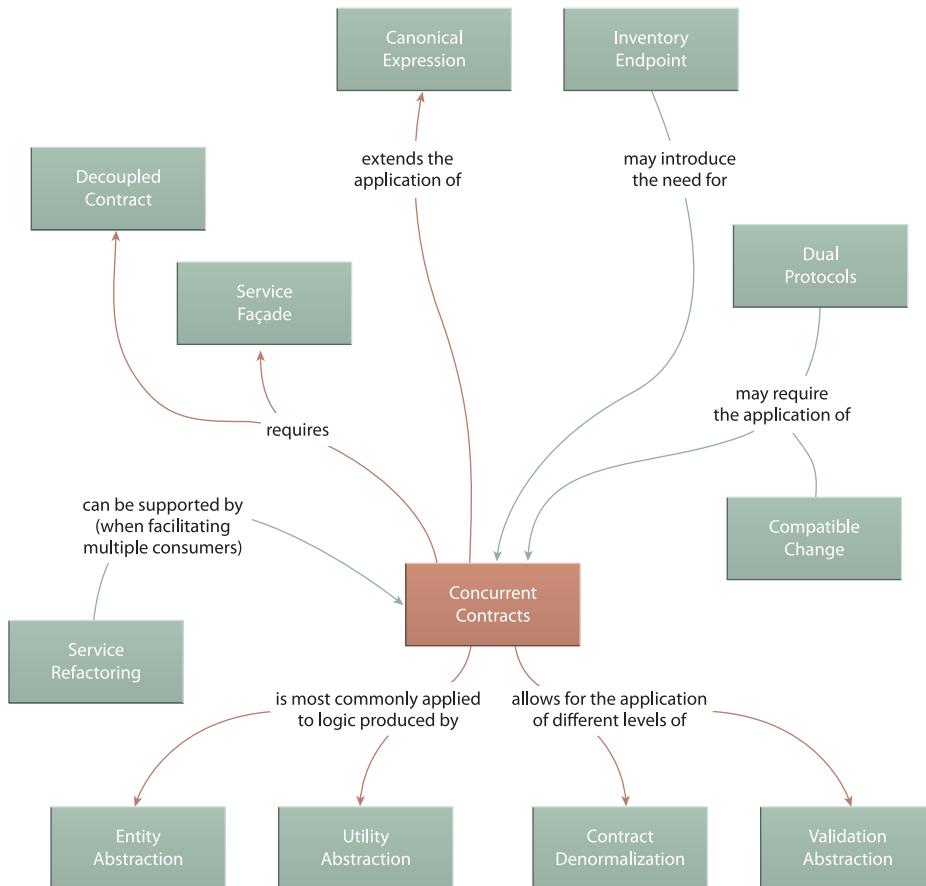
Impacts

From the perspective of the consumer, adding new contracts to the same service is the equivalent of adding new services. This pattern can therefore lead to the bloating of a service inventory and can further confuse consumer designers trying to discover or choose the appropriate variation of a contract. Additionally, new contracts will tend to introduce governance requirements so as to ensure that each establishes a legitimate endpoint.

Relationships

For Concurrent Contracts to be applied, the service contract itself should ideally be fully decoupled from the underlying service logic as per Decoupled Contract (401), and Service Façade (333) should ideally be applied to provide façade logic that supports multiple contracts without the need for redundant service logic.

Both Contract Denormalization (414) and Validation Abstraction (429) are employed to help optimize service contracts in support of the Service Abstraction design principle and also to facilitate different types of service consumers. Concurrent Contracts allows for varying levels of these two patterns to be applied to individual contracts, all for the same service. However, it could turn out that there is less of a need for Contract Denormalization (414) given that denormalized capabilities could simply exist in a different contract (Figure 14.18).

**Figure 14.18**

Concurrent Contracts relates to other contract-related patterns that help support or are affected by the creation of multiple contracts for a single service.

CASE STUDY EXAMPLE

The FRC requires that all new companies in the forestry industry register themselves by completing a pre-defined application. The application form can be filled out by hand and mailed into the FRC office, or it can be completed online via an electronic form submission process.

An Application service is responsible for eventually receiving and processing both types of application data. To accommodate different consumers providing the input, the service is equipped with two different contracts:

- the Application service contract (Example 14.3)
- the EformApplication contract (Example 14.4)

```
<definitions name="Application"
  targetNamespace="http://frc/app/wsdl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:app="http://frc/app/schema/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://frc/app/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ...
  <portType name="AppInt">
    <operation name="Add">
      <input message="tns:AddApp"/>
      <output message="tns:AddAppConfirm"/>
    </operation>
    <operation name="Update">
      <input message="tns:UpdateApp"/>
      <output message="tns:UpdateAppConfirm"/>
    </operation>
    <operation name="Delete">
      <input message="tns:DeleteApp"/>
      <output message="tns:DeleteAppConfirm"/>
    </operation>
    <operation name="Get">
      <input message="tns:GetApp"/>
      <output message="tns:GetAppResults"/>
    </operation>
  </portType>
  ...
</definitions>
```

Example 14.3

The internal Application service contract providing a basic set of operations for the management of application data.

While the internally used Application service provides a relatively standard set of data processing operations, the EformApplication contract is noticeably different. The Delete operation is intentionally omitted, and the base WSDL definition is extended with a policy assertion.

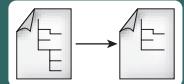
```
<definitions name="EformApplication" ...>
  ...
  <portType name="AppInt">
    <operation name="Add">
      <input message="tns:AddApp" />
      <output message="tns:AddAppConfirm" />
    </operation>
    <operation name="Update">
      <input message="tns:UpdateApp" />
      <output message="tns:UpdateAppConfirm" />
    </operation>
    <operation name="Get">
      <input message="tns:GetApp" />
      <output message="tns:GetAppResults" />
    </operation>
  </portType>
  ...
</definitions>
<wsp:Policy xmlns:wsp="...">
  ...
  <wsp:ExactlyOne>
    <wsp:SpecVersion wsp:Usage="wsp:Required"
      wsp:Preference="10"
      wsp:URI="http://schemas.xmlsoap.org/ws/2004/03/rm" />
    <wsp:SpecVersion wsp:Usage="wsp:Required"
      wsp:Preference="1"
      wsp:URI="http://schemas.xmlsoap.org/ws/2003/02/rm" />
  </wsp:ExactlyOne>
</wsp:Policy>
```

Example 14.4

A second service contract is created specifically for service consumers transmitting application documents electronically. In this version, the Delete operation is removed, and a policy is introduced requiring consumers to support one of two WS-ReliableMessaging specifications.

Validation Abstraction

How can service contracts be designed to more easily adapt to validation logic changes?



Problem	Service contracts that contain detailed validation constraints become more easily invalidated when the rules behind those constraints change.
Solution	Granular validation logic and rules can be abstracted away from the service contract, thereby decreasing constraint granularity and increasing the contract's potential longevity.
Application	Abstracted validation logic and rules need to be moved to the underlying service logic, a different service, a service agent, or elsewhere.
Impacts	This pattern can somewhat decentralize validation logic and can also complicate schema standardization.
Principles	Standardized Service Contract, Service Loose Coupling, Service Abstraction
Architecture	Service

Table 14.5

Profile summary for the Validation Abstraction pattern.

Problem

When building services as Web services, a great deal of validation logic can be expressed using the XML Schema and WS-Policy languages. These standards provide a comprehensive range of features that allow for the definition of very precise and sophisticated validation rules and constraints. By deferring the majority of validation constraints to the service contract, the underlying service logic is alleviated from having to concern itself with the validity and legitimacy of incoming message contents.

However, as part of the technical service contract, this validation logic expresses fixed terms of engagement to which all potential consumer programs need to comply. The day the underlying business rules or requirements (upon which some of the validation constraints may be based) change, it may not be possible to make the corresponding changes to the established contract without releasing a new version. New contract versions introduce governance burden, especially with agnostic services that have many consumers (Figure 14.19).

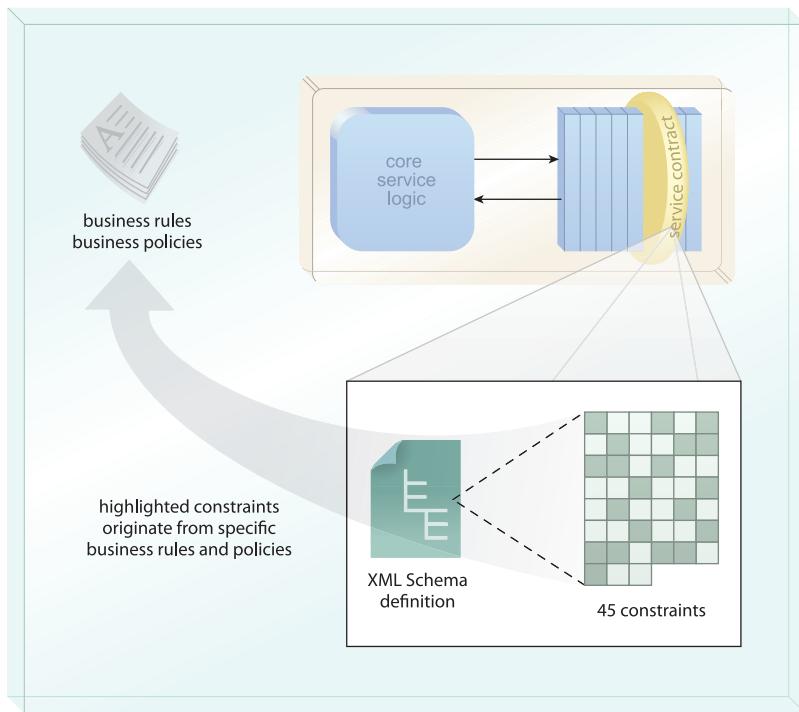


Figure 14.19

Many validation constraints that find their way into contract schemas are tied directly back to business rules and policies that may be subject to change. (In this figure, the dark green squares represent constraints based on business rules and policies.)

Solution

Depending on the nature of the message data being exchanged, there may be opportunities to decrease the constraint granularity of contract capabilities. This leads to a reduction in the quantity and restrictiveness of validation logic embedded in the service contract by deferring select validation constraints elsewhere.

The less validation logic in the contract, the lower the risk of the contract being impacted by overarching business changes. Therefore, the potential longevity of a service contract is extended (Figure 14.20).

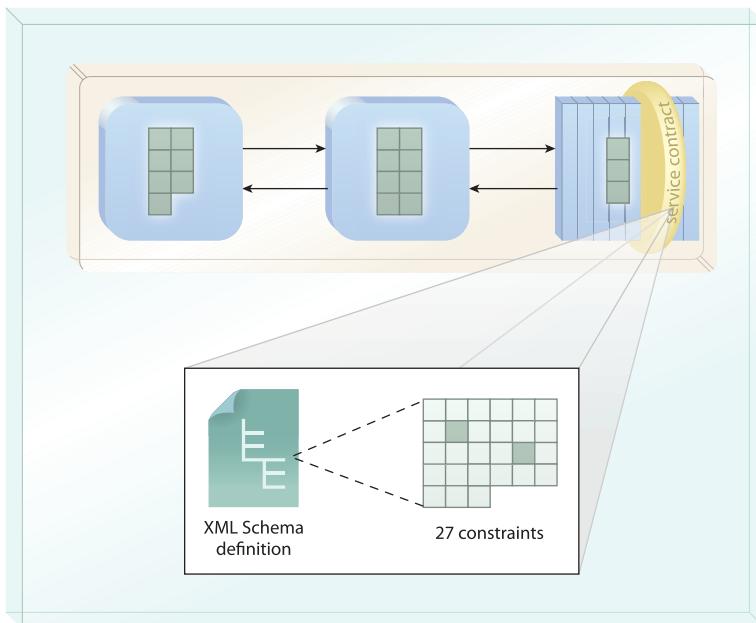


Figure 14.20

By reducing the overall quantity of constraints and especially filtering out those more prone to change, the longevity of a service contract can be extended.

Application

This is an example of a pattern that supports service design but imposes change upon the underlying schema and policy definitions. In other words, it asks that schemas and policies be somewhat more conservatively designed in support of service longevity.

Examples of the types of validation logic that Validation Abstraction tends to target include:

- detailed and granular validation constraints that express very specific conditions
- constraints based upon precise value characteristics (such as null values or the minimum or maximum allowable length of a document value)
- constraints based on embedded enumeration values or code lists
- policy expressions that define specific properties or behaviors derived from business rules

Another area in which this pattern can be effectively applied is the actual typing of message data. Instead of tuning data types to current document definition requirements, an approach can be adopted whereby more lenient data types are employed. This especially affects the simple types used within XML schemas to represent specific document values. Furthermore, attachments can even be utilized to bypass contract-level typing altogether.

As with the other deferred constraints, the actual validation of the affected values occurs within the underlying solution logic. For validation and policy-based constraints that still need to be communicated to consumer program designers, descriptions of the constraints can be added to supplemental service contract documentation, such as the SLA.

Impacts

One of the benefits of decoupled service contracts is that they provide a central location for the placement of validation logic. All of the constraints messages need to comply to can be enforced at the outer rim of the service boundary so that only valid messages make their way through to the underlying service logic.

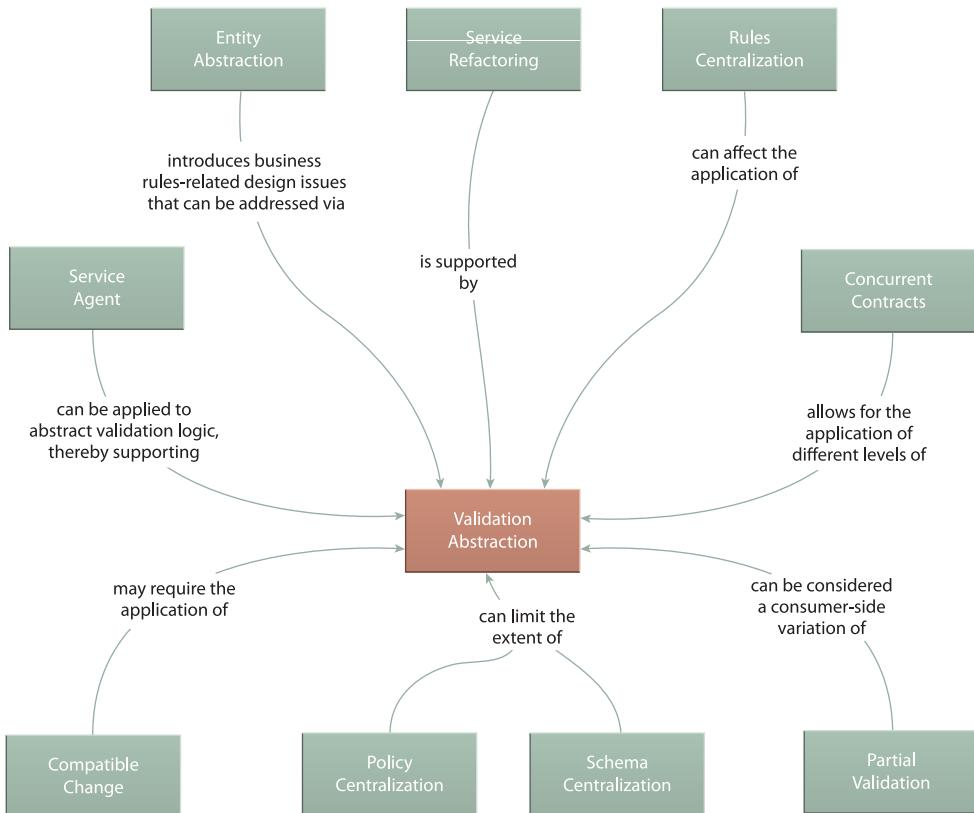
Applying this pattern removes validation logic from the contract layer, decentralizing it and thereby requiring it to be maintained in different locations. Although this increases maintenance effort, this increase is generally not equivalent to the governance impact of having to introduce a new version of a service contract or a new capability.

Relationships

When applying Validation Abstraction, it is important to take Policy Centralization (207) and Schema Centralization (200) into account because when contracts use centralized policies or schemas, their validation logic may not be able to be abstracted as much as this pattern advocates.

One of the major types of logic this pattern looks to remove or hide from the service contract is business rules. Therefore, Rules Centralization (216) is related, as is Entity Abstraction (175) due to the fact that it produces services that encapsulate business entity logic (which generally includes associated business rules) and also posses contracts with multiple capabilities that may be affected by this pattern.

Where exactly abstracted validation logic goes is not dictated by this pattern, which is why Service Agent (543) and Rules Centralization (216) provide possible options. When applying Concurrent Contracts (421), this pattern can be used to customize each contract to facilitate different consumer requirements.

**Figure 14.21**

Validation Abstraction's goal of streamlining service contract content can directly tie into the application of some patterns, while also meeting resistance from others.

CASE STUDY EXAMPLE

The case study example for Concurrent Contracts (421) introduced portions of the WSDL definition for the EformApplication service contract, including the following construct for the Add operation:

```
<operation name="Add">
  <input message="tns:AddApp" />
  <output message="tns:AddAppConfirm" />
</operation>
```

The existing validation logic for externally received application logic is comprised of the following rules:

- Applications must be accompanied by an ID value auto-generated by the client user-interface used by the person to fill out and then submit the application electronically on behalf of the company. This ID must consist of two sets of three digits separated by a hyphen.
- The name of the company submitting the application. The name value is limited to 100 characters.
- The federal tax number of the company. This value is limited to 12 digits.
- The company's FRC classification, as selected via a pre-populated drop-down list by the person filling out the online application.
- Additional remarks provided by the person completing the application via an open, multi-line text field.

The original XML Schema types associated with the AddApp message mirror these validation rules, as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://frc/applications"
  xmlns="http://frc/genproc">
  <xsd:element name="submission" type="AppType" />
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="submission" type="AppType"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="AppType" >
    <xsd:sequence>
      <xsd:element name="AppID" type="AppIDType" />
      <xsd:element name="Name" type="NameType" />
      <xsd:element name="TaxNum" type="TaxNumType" />
      <xsd:element name="Class" type="ClassType" />
      <xsd:element name="Comments" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="AppIDType" >
    <xsd:restriction base="xsd:string" >
      <xsd:pattern value="\d{3}\-\d{3}\\" />
```

```
</xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="NameType">
    <xsd:restriction base="xsd:string">
        <xsd:length value="100"/>
    </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="TaxNumType">
    <xsd:restriction base="xsd:integer">
        <xsd:totalDigits value="12"/>
    </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="ClassType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="CDA"/>
        <xsd:enumeration value="TRU"/>
        <xsd:enumeration value="CMI"/>
        <xsd:enumeration value="TER"/>
        <xsd:enumeration value="LBR"/>
        <xsd:enumeration value="LBA"/>
        <xsd:enumeration value="MAG"/>
        <xsd:enumeration value="IST"/>
        <xsd:enumeration value="GER"/>
    </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Example 14.5

The original schema definition for the EformApplication service contract.

It was initially considered reasonable for the AddApp operation to have a relatively fine-grained level of constraint coupling because the application documents received by this service were submitted only by Web client logic also developed by the FRC. If the business validation rules changed, then both the Web forms and the schema types for the EformApplication service contract were also changed correspondingly.

However, a recent requirement emerged to allow non-Web client-based programs to also submit and retrieve application data. Security policies do not allow for another external endpoint to be published for application processing, which means that the EformApplication contract now needs to accommodate data exchanges with both Web clients and B2B-style partner service consumers.

To date, the EformApplication service has been in production for five months during which one classification code had to be removed and two new codes had to be added.

Once external partner consumer programs bind to the published EformApplication contract, it becomes cumbersome and risky to change validation constraints, such as embedded code lists. The team responsible for this service decides to loosen the validation rules in the contract by replacing the enumerated class code list with an allowable input string of three characters, as shown here:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://frc/applications"
  xmlns="http://frc/genproc">
  <xsd:element name="submission" type="AppType"/>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="submission" type="AppType"
        maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="AppType">
    <xsd:sequence>
      <xsd:element name="AppID" type="AppIDType"/>
      <xsd:element name="Name" type="NameType"/>
      <xsd:element name="TaxNum" type="TaxNumType"/>
      <xsd:element name="Class" type="ClassType"/>
      <xsd:element name="Comments" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="AppIDType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}\-\d{3}\\"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="NameType">
    <xsd:restriction base="xsd:string">
      <xsd:length value="100"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="TaxNumType">
    <xsd:restriction base="xsd:integer">
      <xsd:totalDigits value="12"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="ClassType">
```

```
<xsd:restriction base="xsd:string">
  <xsd:length value="3" />
</xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Example 14.6

The revised XML Schema definition with coarser-grained constraint granularity due to the abstraction of classification code validation.

This page intentionally left blank

Chapter 15



Legacy Encapsulation Patterns

Legacy Wrapper

Multi-Channel Endpoint

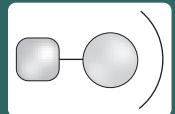
File Gateway

Introducing a collection of standardized service endpoints into an established IT enterprise will almost always result in a need to manage the marriage of service-orientation with legacy encapsulation. This chapter provides a set of patterns dedicated to addressing common challenges with service encapsulation of legacy systems and environments.

Legacy Wrapper (441) establishes the fundamental concept of wrapping proprietary legacy APIs with a standardized service contract, while Multi-Channel Endpoint (451) builds on this concept to introduce a service that decouples legacy systems from delivery channel-specific programs. File Gateway (457) further provides bridging logic for services that need to encapsulate and interact with legacy systems that produce flat files.

Legacy Wrapper

By Thomas Erl, Satadru Roy



How can wrapper services with non-standard contracts be prevented from spreading indirect consumer-to-implementation coupling?

Problem	Wrapper services required to encapsulate legacy logic are often forced to introduce a non-standard service contract with high technology coupling requirements, resulting in a proliferation of implementation coupling throughout all service consumer programs.
Solution	The non-standard wrapper service can be replaced by or further wrapped with a standardized service contract that extracts, encapsulates, and possibly eliminates legacy technical details from the contract.
Application	A custom service contract and required service logic need to be developed to represent the proprietary legacy interface.
Impacts	The introduction of an additional service adds a layer of processing and associated performance overhead.
Principles	Standardized Service Contract, Service Loose Coupling, Service Abstraction
Architecture	Service

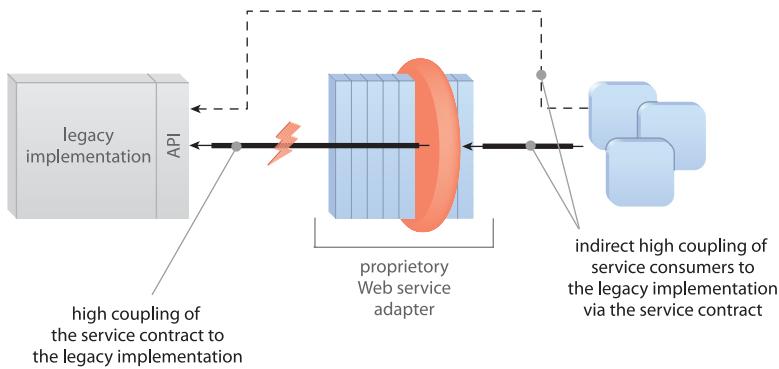
Table 15.1

Profile summary for the Legacy Wrapper pattern.

Problem

Legacy systems must often be encapsulated by services established by proprietary component APIs or Web service adapter products. The resulting technical interface is frequently fixed and non-customizable. Because the contract is pre-determined by the product vendor or constrained by legacy component APIs, it is not compliant with contract design standards applied to a given service inventory.

Furthermore, the nature of API and Web service adapter contracts is often such that they contain embedded, implementation-specific (and sometimes technology-specific) details. This imposes the corresponding forms of implementation and technology coupling upon all service consumers (Figure 15.1).

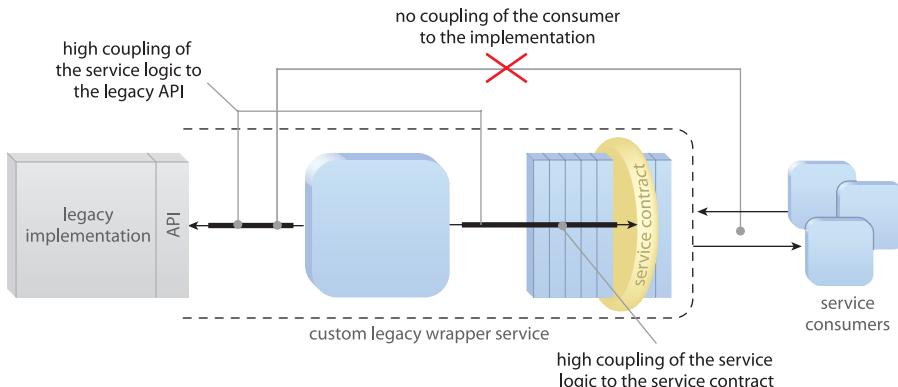
**Figure 15.1**

High contract-to-logic coupling resulting in high implementation coupling by service consumers.

Solution

Although a legacy system API or a Web service adapter will expose an official, generic entry point into legacy system logic, it is often unadvisable to classify such an endpoint as an official member of a service inventory. Instead, it can be safer to view legacy APIs and Web service adapters as extensions of the legacy environment providing just another proprietary interface that is available for service encapsulation.

This perspective allows for the creation of a standardized legacy wrapper service that expresses legacy function in a standardized manner. The result is a design that enables the full abstraction of proprietary legacy characteristics (Figure 15.2), which provides the freedom of evolving or replacing the legacy system with minimal impact on existing service consumers.

**Figure 15.2**

Tight coupling of the service logic to both the legacy API and the service contract alleviates service consumers from implementation coupling.

Application

The application of this pattern is typically associated with the introduction of a new service contract. However, when wrapping only the parts of a legacy resource that fall within a pre-defined service boundary, this pattern may result in just the addition of a new capability to an existing service contract.

Either way, the wrapper service (or capability) will typically contain logic that performs transformation between its standardized contract and the native legacy interface. Often, this form of transformation is accomplished by eliminating and encapsulating technical information, as follows:

- *Eliminating Technical Information* – Often legacy input and output data contain highly proprietary characteristics, such as message correlation IDs, error codes, audit information, etc. Many of these details can be removed from the wrapper contract through additional internal transformation. For example, error codes could be translated to SOAP faults, and message correlation IDs can be generated within the wrapper service implementation. In the latter case, the consumers can communicate with the wrapper service over a blocking communication protocol such as HTTP and hence would not need to know about correlation IDs.
- *Encapsulating Technical Information* – When service consumers still need to pass on legacy-specific data (such as audit-related information) the message exchanged by the legacy wrapper contract can be designed to partition standardized business data from proprietary legacy data into body and header sections, respectively. In this case, both the legacy wrapper service and its consumer will need to carry out additional processing to assemble and extract data from the header and body sections of incoming and outgoing messages.

The former approach usually results in a utility service whereas the latter option will tend to add the wrapper logic as an extension of a business service. It can be beneficial to establish a sole utility wrapper service for a legacy system so that all required transformation logic is centralized within that service's underlying logic. If capabilities from multiple services each access the native APIs or Web service adapter interfaces, the necessary transformation logic will need to be distributed (decentralized). If a point in time arrives where the legacy system is replaced with newer technology, it will impact multiple services.

NOTE

Some ERP environments allow for the customization of local APIs but still insist on auto-generating Web service contracts. When building services as Web services, these types of environments may still warrant encapsulation via a separate, standardized Web service. However, be sure to first explore any API customization features. Sometimes it is possible to customize a native ERP object or API to such an extent that actual contract design standards can be applied. If the API is standardized, then the auto-generated Web service contract also may be standardized because it will likely mirror the API.

Impacts

Adding a new wrapper layer introduces performance overhead associated with the additional service invocation and data transformation requirements.

Also, expecting a legacy wrapper contract alone to fully shield consumers from being affected from when underlying legacy systems are changed or replaced can be unrealistic. When new systems or resources are introduced into a service architecture, the overall behavior of the service may be impacted, even when the contract remains the same. In this case, it may be required to further supplement the service logic with additional functions that compensate for any potential negative effects these behavioral changes may have on consumers. Service Façade (333) is often used for this purpose.

Relationships

Legacy Wrapper makes legacy resources accessible on an inter-service basis. It can therefore be part of any service capability that requires legacy functionality. Entity and utility services are the most common candidates because they tend to encapsulate logic that represents either fixed business-centric boundaries or technology resources. Therefore, this pattern is often applied in conjunction with Entity Abstraction (175) and Utility Abstraction (168).

Patterns that often introduce proprietary products or out-of-the-box services, such as Rules Centralization (216), also may end up having to rely on Legacy Wrapper to make their services part of a federated service inventory. Service Data Replication (350) can be combined with this pattern to provide access to replicated proprietary repositories, and File Gateway (457) may be applied to supplement the wrapper contract with specialized internal legacy encapsulation logic.

Because of the broker-related responsibilities that a legacy wrapper service will generally need to assume, it is further expected that it will require the application of Data Format Transformation (681) or Data Model Transformation (671), and possibly Protocol Bridging (687).

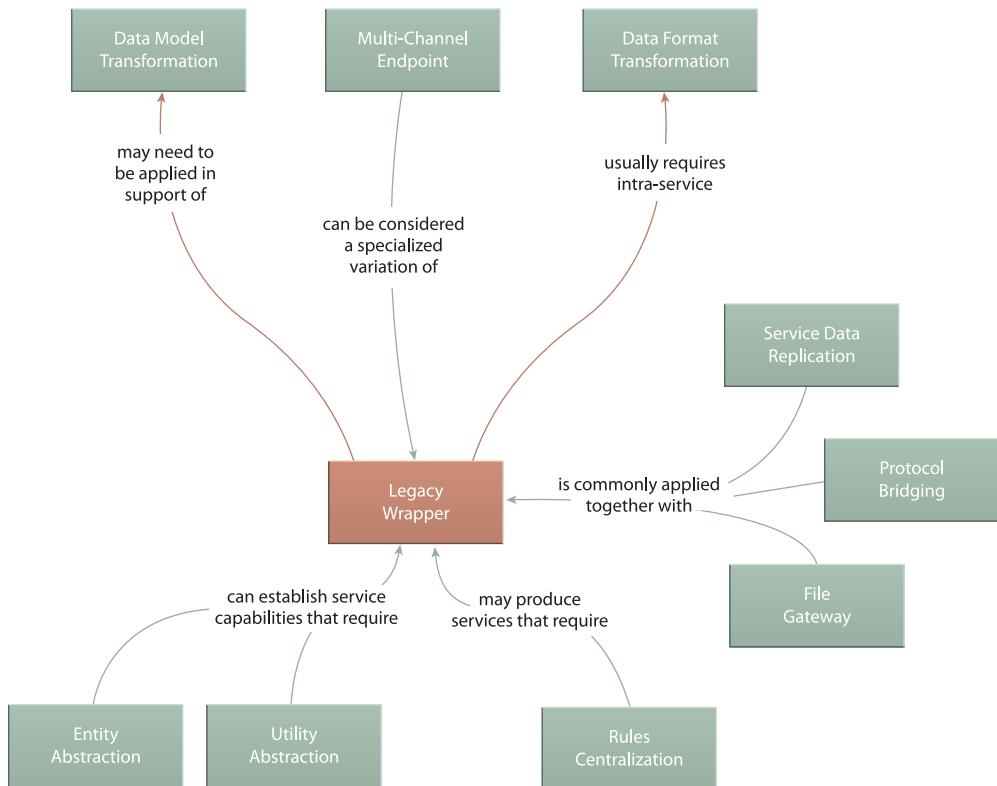


Figure 15.3

Legacy Wrapper provides a convenient means of bringing legacy logic into a service inventory without compromising the integrity of the service contracts. It therefore is of interest to any pattern that may require access to legacy resources.

Legacy Wrapper is therefore frequently applied together with Enterprise Service Bus (704) in order to leverage the broker capabilities natively provided by ESB platforms.

CASE STUDY EXAMPLE

Subsequent to the implementation of the Appealed Assessments service (described in the Service Façade (333) case study example), the Data Controller component is modified to no longer provide access to one of the four data repositories. Instead, architects plan to add the lost data access functionality to the Appealed Assessments service's Data Relayer component. This would make that component less of a service façade, while continuing to preserve the functionality promised by the Appealed Assessments service contract.

However, an idea for another option emerges. The repository in question is part of a legacy environment (called MainAST103) that runs on mainframe technologies and limits data access to a generic CICS API but also provides the option of auto-generating a wrapper Web service interface to mirror this API. At first this seems interesting, but when the auto-generated WSDL (with embedded XML Schema content) is studied, the FRC architects want no part of it.

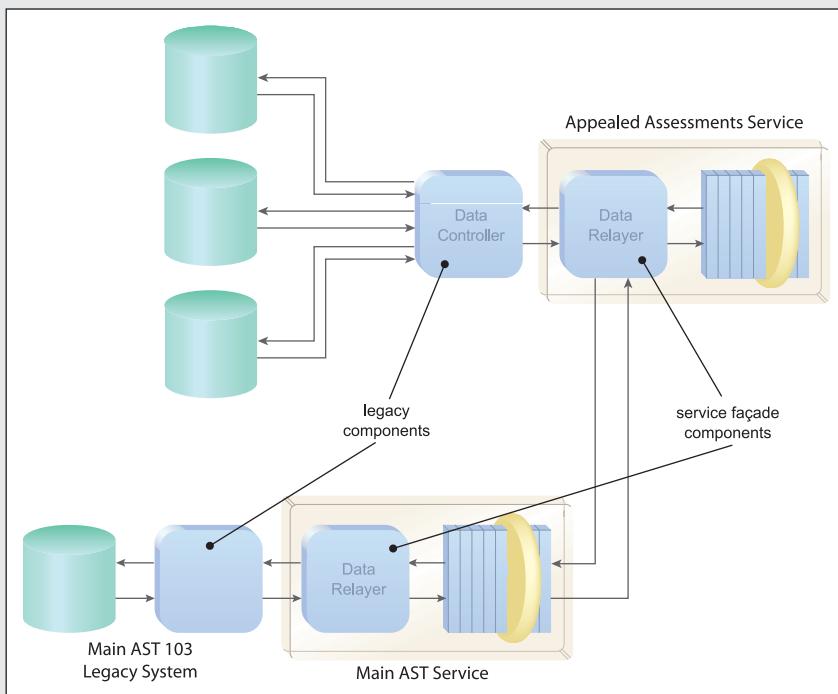


Figure 15.4

The revised Appealed Assessments service architecture.

Instead, they opt to create a new MainAST utility service (Figure 15.4) to “wrap” this legacy mainframe system and associated API. The functional context of this utility service is limited to this one CICS API accessed via MQ messaging, and its core service logic is dedicated to converting requests, responses, and data to and from the standardized Web service contract models and the native, proprietary legacy API formats.

Provided here are some fragments of the input and output messages exchanged by the CICS API:

```
000100      01 ASSESSMENT-APPEAL-REQUEST .  
000200      05 ASSESSMENT-APPEAL-CONTROL-DATA .  
000300      10 SECURITY-TOKEN      PIC X(10) .  
000500      10 MESSAGE-ID        PIC X(20) .  
000600      10 MESSAGE-CORR-ID    PIC X(20) .  
000700      05 ASSESSMENT-APPEAL -DATA .  
000800      10 AGENCY-ID         PIC X(20) .  
000900      10 ASSESSMENT-NUMBER   PIC X(10) .  
001000      10 APPEAL-REASON-CODE PIC X(10) .  
001100      10 APPEAL-DATE       PIC X(08) .
```

Example 15.1

A fragment of the COBOL COPYBOOK description for the input message of the legacy CICS API. Note that because communication with the MainAST 103 legacy program is carried out through MQ-based messaging, technical information such as security tokens, message IDs and correlation IDs are expected and used for authorization and message correlation purposes.

```
000100      01 ASSESSMENT-APPEAL-RESPONSE .  
000200      05 ASSESSMENT-APPEAL-RESPONSE-CONTROL-DATA .  
000300      10 MESSAGE-ID          PIC X(20) .  
000400      10 MESSAGE-CORR-ID    PIC X(20) .  
000700      05 ASSESSMENT-APPEAL-RESPONSE-DATA .  
000800      10 RETURN-CODE        PIC 9(2) .  
000900      10 ERROR-TYPE         PIC X(2) .  
000900      10 ERROR-CODE-DESC    PIC X(20) .  
001000      10 APPEAL-RESULT-CODE PIC X(10) .
```

Example 15.2

A fragment of the COBOL COPYBOOK for the response message. This response message also contains technical information, such as message and correlation IDs. As is typical with mainframe-based APIs, the result of the transaction invocation (and possible errors) are communicated through the return code, error code, error descriptions, etc.

Based on some preliminary analysis, FRC architects come up with the following WSDL definition for the legacy wrapper service contract:

```
<definitions targetNamespace="http://frc/assessments/appeals">
  <types>
    <xsd:schema ...>
      <xsd:element name="AppealAssessmentRequestData_Type">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="agencyId" type="string"/>
            <xsd:element name="assessmentNumber" type="int"/>
            <xsd:element name="appealReasonCode" type="string"/>
            <xsd:element name="appealDate" type="Date"/>
            <!-- Other types -->
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="AppealAssessmentResponseData_Type">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="appealResultCode" type="string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="InsufficientAppealInformation">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="description" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </types>
  <message name="appealAssessmentInputMessage">
    <part name="input" element=
      "tns:AppealAssessmentRequestData_Type"/>
  </message>
  <message name="appealAssessmentOutputMessage">
    <part name="output" element=
      "tns:AppealAssessmentResponseData_Type"/>
  </message>
  <message name="insufficientAppealInformationException">
    <part name="fault" element=
      "tns:InsufficientAppealInformation"/>
  </message>
<portType name="AppealAssessment">
```

```
<operation name="appealAssessment">
    <input message="tns:appealAssessmentInputMessage" />
    <output message="tns:appealAssessmentOutputMessage" />
    <fault name="fault" message=
        "tns:insufficientAppealInformationException" />
</operation>
</portType>
</definitions>
```

Example 15.3

The WSDL definition for the MainAST legacy wrapper service.

Notice how in Example 15.3 the request and response message types do not contain any reference to legacy technical information, even though all of the proprietary technical details are still required by the MainAST system. However, this requires that FRC architects still need to find a means of supplying that information to the legacy system without exposing the details to the service consumer, which is why they apply the eliminate and encapsulate steps, as follows:

Technical details are removed because the legacy wrapper service is designed to handle low-level MQ request and response messages with the MainAST 103 system on its own. For example, error code information is eliminated because the wrapper service can simply use the fault message defined in the abstract WSDL description. Within the service logic, error codes returned by the legacy system are translated to an `InsufficientAppealInformationException` message.

However, it becomes evident that information needed for service authorization cannot be eliminated altogether. The FRC team decides that these details can be packaged into SOAP headers instead, as shown in the following WSDL code:

```
<definitions ...>
    <types>
        <!-- type definition for the legacy header data -->
        <element name="securityheader">
            <complexType>
                <sequence>
                    <element name="securetoken" type="xsd:string" />
                </sequence>
            </complexType>
        </element>
        ...
    </types>
    ...

```

```
<!-- message definition for the legacy header -->
<message name="appealAssessmentHeaderData">
    <part name="header" element="securityheader"/>
    <!-- Other technical information can be included here -->
</message>
...
<!-- SOAP header declaration in the WSDL binding -->
<binding name="AppealAssessmentBinding" type=
    "tns:AppealAssessmentService">
    <soap:binding style="document" transport=
        "http://schemas.xmlsoap.org/soap/http"/>
    <operation name="appealAssessment">
        <soap:operation soapAction="" />
        <input>
            <soap:body message=
                "tns:appealAssessmentInputMessage" use="literal"/>
            <soap:header message="appealAssessmentHeaderData"
                part="header" use="literal"/>
        </input>
        <output>
            <soap:body message=
                "appealAssessmentOutputMessage" use="literal"/>
        </output>
    </operation>
</binding>
...
</definitions>
```

Example 15.4

These code fragments show how legacy details can be isolated inside SOAP headers. Note the SOAP header is considered implicit because it is not referred to in the `portType` construct.

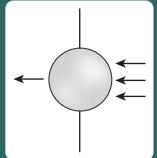
From a service consumer's point of view, any out-of-band, technical information can now be communicated via the SOAP headers, while the core business service contract remains unaffected.

Although the need for this service arose out of the data access requirements of the Appealed Assessments service, the project team responsible for delivering the MainAST service is given the go-ahead to add further capabilities so that it can provide a range of generic functions that fully leverage the underlying legacy API (and go well beyond the functionality required by the Appealed Assessments service).

Multi-Channel Endpoint

By Satadru Roy

How can legacy logic fragmented and duplicated for different delivery channels be centrally consolidated?



Problem	Legacy systems custom-built for specific delivery channels (mobile phone, desktop, kiosk, etc.) result in redundancy and application silos when multiple channels need to be supported, thereby making these systems burdensome to govern and difficult to federate.
Solution	An intermediary service is designed to encapsulate channel-specific legacy systems and expose a single standardized contract for multiple channel-specific consumers.
Application	The service established by this pattern will require significant processing and workflow logic to support multiple channels while also coordinating interaction with multiple backend legacy systems.
Impacts	The endpoint processing logic established by this pattern often introduces the need for infrastructure upgrades and orchestration-capable middleware and may turn into a performance bottleneck.
Principles	Service Loose Coupling, Service Reusability
Architecture	Service

Table 15.2

Profile summary for the Multi-Channel Endpoint pattern.

Problem

Some solutions need to exchange information with data sources or other systems via different delivery channels. For example, a banking solution may need to receive the same account data from a desktop client, a self-service Internet browser, a call center, a kiosk, an ATM, or even a mobile device. Each of these delivery channels will have its own communications and data representation requirements.

Traditional legacy systems were commonly customized to support specific channels. When new channels need to be supported, new systems were developed, and/or existing

systems were integrated. As a result, legacy multi-channel environments led to multiple silos and duplicated islands of data and functionality (Figure 15.5).

This, in turn, led to inconsistency in how systems delivered the same functionality across different implementations and further helped proliferate disparate data sources and tightly coupled consumer-to-system connections.

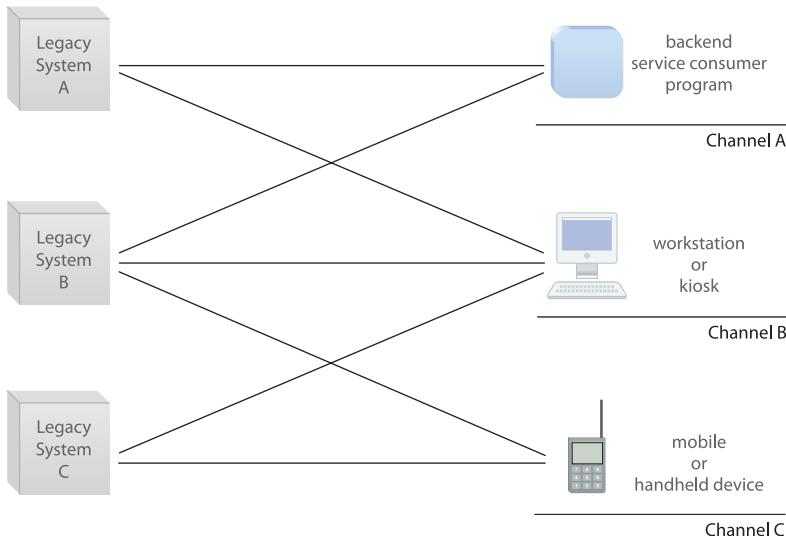


Figure 15.5

In legacy environments, channel-based solutions (right) were often forced to integrate directly with different legacy systems (left).

Solution

A service is introduced to decouple both modern solutions and legacy systems from multiple channels. The architecture for this multi-channel service is designed so that it exposes a single, standardized contract to the channel providers, while containing all of the necessary transformation, workflow, and routing logic to ensure that data received via different channels is processed and relayed to the appropriate backend systems (Figure 15.6).

The abstraction achieved by this service insulates legacy systems and other services from the complexities and disparity of multi-channel data exchange and further insulates the channel providers from forming negative types of coupling.

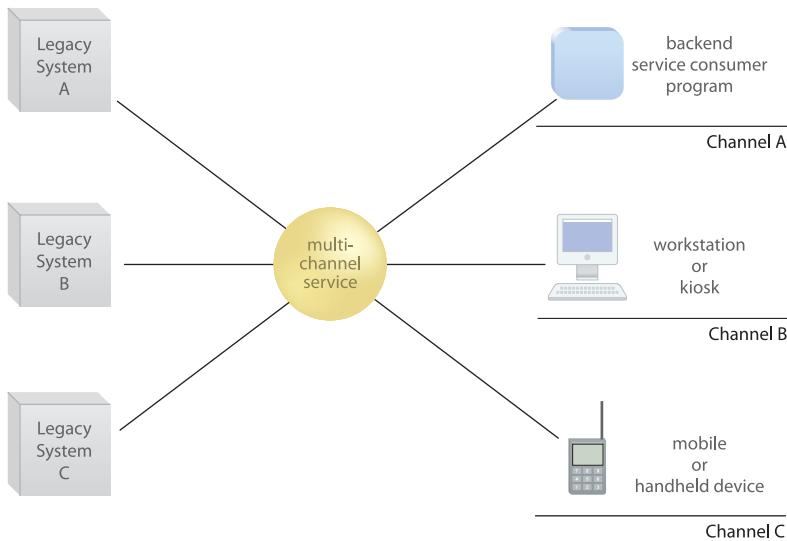


Figure 15.6

The multi-channel service acts as the central contact point for different channel-based solutions (right) and legacy systems (left).

Application

A multi-channel service essentially encapsulates process logic associated with common tasks. Its primary responsibility is to ensure that these tasks get carried out consistently, regardless of where (or which channel) input data comes from. Designing multi-channel services requires careful analysis of the data offered over the supported channels to ensure that the single service contract can serve the (often disparate) requirements of different channel-based consumers.

This pattern is based on many of the same concepts as Legacy Wrapper (441), except that it is specific to supporting multiple delivery channels. Multi-Channel Endpoint is therefore commonly applied with industry-standard technologies, such as those that relate to Web services and REST services, so as to support as wide a range of potential channels as possible.

When building these services as Web services, SOAP headers can be used to establish a separation of channel-specific information in order to avoid convoluted or unreasonably bloated contract content. Additionally, the legacy channel-specific systems themselves may need to be augmented to work with the multi-channel service. This change may especially be significant when this pattern is combined with Orchestration (701), in which case legacy systems may find themselves as part of larger orchestrated business processes.

Multi-channel services often warrant classification as a distinct service model that is a combination of task and utility service. They are similar to task services in that they encapsulate logic specific to business processes, but unlike task services, their reuse potential is higher because the business processes in question are common across multiple channels.

Because they are often required to provide complex broker, workflow, and routing logic, it is further typical for multi-channel services to be hosted on ESB and orchestration platforms. Some ESB products even provide legacy system adapters with built-in multi-channel support.

Impacts

Even though this pattern helps decouple traditionally tightly-bound clients and systems, it can introduce the need for services that themselves need to integrate and become tightly coupled to various backend legacy environments and resources.

Due to the highly processing-intensive nature of some multi-channel services, this pattern can lead to the need for various infrastructure upgrades, including orchestration engines and message brokers. Depending on the amount of concurrent data exchanged, multi-channel services can easily become performance bottlenecks due to the quantity of logic they are generally required to centralize.

Relationships

In many ways, Legacy Wrapper (441) acts as a root pattern for Multi-Channel Endpoint in that it establishes the concept of abstraction and decoupling legacy resources. As shown in Figure 15.7, multi-channel services are almost always required to carry out various forms of runtime transformation when processing incoming channel data, which is why this pattern often needs to rely on Service Broker (707) patterns, such as Protocol Bridging (687), Data Format Transformation (681), and Data Model Transformation (671).

Composition Autonomy (616) and Redundant Implementation (345) can be further applied to help scale and support multi-channel services in high volume environments. Service Façade (333) is also useful for abstracting routing, workflow, and broker logic within the internal service architecture.

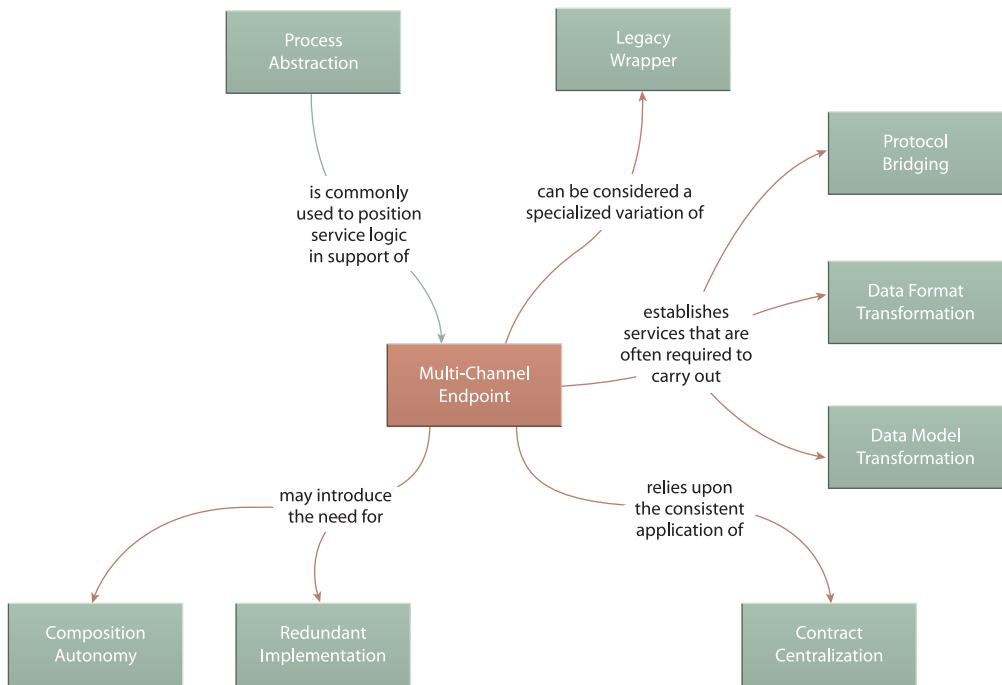


Figure 15.7

Because of its similarity to Inventory Endpoint (260), Multi-Channel Endpoint relates to several of the same patterns.

NOTE

Multi-Channel Endpoint can be viewed as an extension of the Multi-Channel Access strategy described in *Understanding SOA with Web Services* (Newcomer, Lomow). Whereas Multi-Channel Access emphasizes the reusability aspect of services across channels, Multi-Channel Endpoint focuses on centralized brokering and workflow abstraction aspects critical to the implementation of channel-independent services.

CASE STUDY EXAMPLE

Whenever a lumber company decides to appeal an FRC policy violation assessment, it needs to initiate a separate appeals process for which it also is required to pay a new fee.

The FRC further offers an EDI system that can be accessed by authorized companies (or their attorneys) to submit these payments. The FRC also operates a call center that takes calls from company accounting departments through which payments can also be made. In this case, the call center representative collects the payment information over the phone and enters it into an internal system.

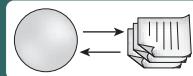
Each method of payment translates into a different delivery channel for payment data. Historically, these channels were independently supported by different systems. The call center system is a custom-developed application that integrates with the central accounting system, whereas the EDI interface is essentially a dated B2B system that has been in place for 10 years and acts as a wrapper for a legacy database.

Payment data is collected in different formats and stored in different repositories. Once a month, batch exports are carried out to extract payment data from the B2B database into the accounting system.

With talk of allowing FRC field agents to collect payments remotely and submit them via mobile devices, architects want to take steps to avoid introducing yet another application and data silo. They decide to introduce an intermediary processing layer based on the multi-channel service. This layer splits the backend call center application and the EDI environment away from the front-end call center application and external EDI consumers.

File Gateway

By Satadru Roy



How can service logic interact with legacy systems that can only share information by exchanging files?

Problem	Data records contained in flat files produced by a legacy system need to be processed individually by service logic, but legacy systems are not capable of directly invoking services. Conversely, service logic may need to produce information for the legacy system, but building file creation and transfer functionality into the service can result in an inflexible design.
Solution	Intermediary two-way file processing logic is positioned between the legacy system and the service.
Application	For inbound data the file gateway processing logic can detect file drops and leverage available broker features to perform Data Model Transformation (671) and Data Format Transformation (681). On the outbound side, this logic intercepts information produced by services and packages them (with possible transformation) into new or existing files for consumption by the legacy system.
Impacts	The type of logic provided by this pattern is unsuitable when immediate replies are required by either service or legacy system. Deployment and governance of two-way file processing logic can further add to operational complexity and may require specialized administration skills.
Principles	Service Loose Coupling
Architecture	Service

Table 15.3

Profile summary for the File Gateway pattern.

Problem

Quite often, service-enabled applications need to process information that is produced only periodically by legacy systems as flat files that can contain a large number of data records separated by delimiters. Any application that needs to access the data in these files will usually be required to iterate through each record to perform necessary calculations.

However, such applications also have to take on the additional responsibility of polling directories to check when the files are actually created and released by the legacy system. Once detected, these files then may need to be parsed, moved (or removed), renamed, transformed, and archived before processing can be considered successful. This has traditionally resulted in overly complex and inflexible data access architectures that revolve around file drop polling and file format-related processing.

The limitations of this architecture further compound when this form of processing needs to be encapsulated by a service. Legacy systems are designed with the assumption that other legacy consumers are accessing the flat files they produce and therefore have no concept of a service, let alone a means of invoking services (as depicted in Figure 15.8).

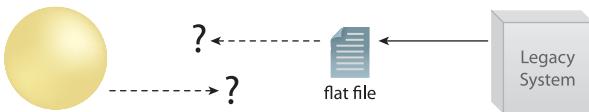


Figure 15.8

The flat files produced by legacy system are intended for customized legacy consumers, not for services.

Solution

Service-friendly gateway logic is introduced, capable of handling file detection and processing and subsequent communication with services. This allows services to consume flat files as legacy consumer systems would. It further introduces the opportunity to optimize this logic to improve upon any existing limitations in the legacy environment (Figure 15.9).

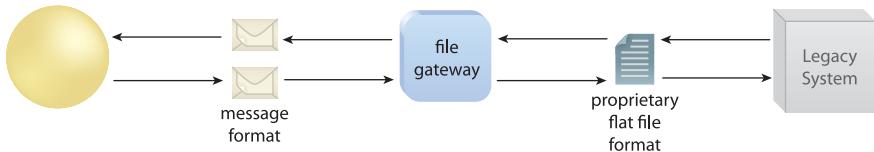


Figure 15.9

File gateway logic acts as a mediator between a service and a flat-file-based legacy system.

Application

The gateway logic introduced by this pattern is most commonly wrapped in a separate utility component positioned independently from business services and is therefore capable of acting as an intermediary between legacy systems and services requiring access to the

legacy data. However, the option also exists to limit this logic to individual service architectures.

This file processing logic will typically be comprised of routines that carry out the following tasks:

- File drop polling with configurable polling cycles.
- File and delimiter parsing and data retrieval with configurable parameters that determine how many records should be read and processed at one time.
- File data transformation, as per Data Format Transformation (681) and possible Data Model Transformation (671).
- Service invocation and file data transfer to services.
- File cleanup, renaming, and/or archiving.

Note that making the polling frequency and number of records processed at a time configurable allows the file processing logic to control the volume of messages sent to a service over a period of time. If a given service takes longer than usual to process a batch of records, these two settings can be tweaked to allow the service to catch up and, similarly, the rate can be increased if the service is processing incoming messages quicker than they are generated.

When services need to share data with legacy systems only capable of receiving flat files, this logic can act as the recipient and broker of messages responsible for transforming them into the legacy format. For example, for the first batch of incoming data the file gateway component can write records out to a flat file with special character-based delimiters separating each data record from another. Subsequent incoming data could then be appended to this file until a pre-specified file size limit is reached, after which it creates a new file and repeats this process.

Impacts

Most of the impacts of this pattern are related to the impacts of file transfer in general. Because the process of transferring files is inherently asynchronous, it naturally introduces (often significant) latency to just about any data exchange scenario.

It can further be challenging to position file gateway components as reusable utility services due to the frequent need to configure the parameters of file transfer and processing for each service-to-legacy system file transfer.

Careful planning for management, administration, and monitoring is also required to maintain expectations related to message processing volumes and performance.

Relationships

As explained previously, File Gateway commonly introduces utility services that depend on Data Format Transformation (681) and possible Data Model Transformation (671) to perform the conversion logic necessary for flat file data to be packaged and sent to services.

When viewing file transfer itself as a communications protocol, this pattern can be viewed as a variation of Protocol Bridging (687) as it essentially overcomes disparity in legacy and service communication platforms.

The application of Legacy Wrapper (441) may require File Gateway when a legacy system producing flat files needs to be wrapped by a standardized service contract and Service Agent (543), Service Callback (566), and Event-Driven Messaging (599) can be further applied to establish more sophisticated processing extensions around the file gateway logic.

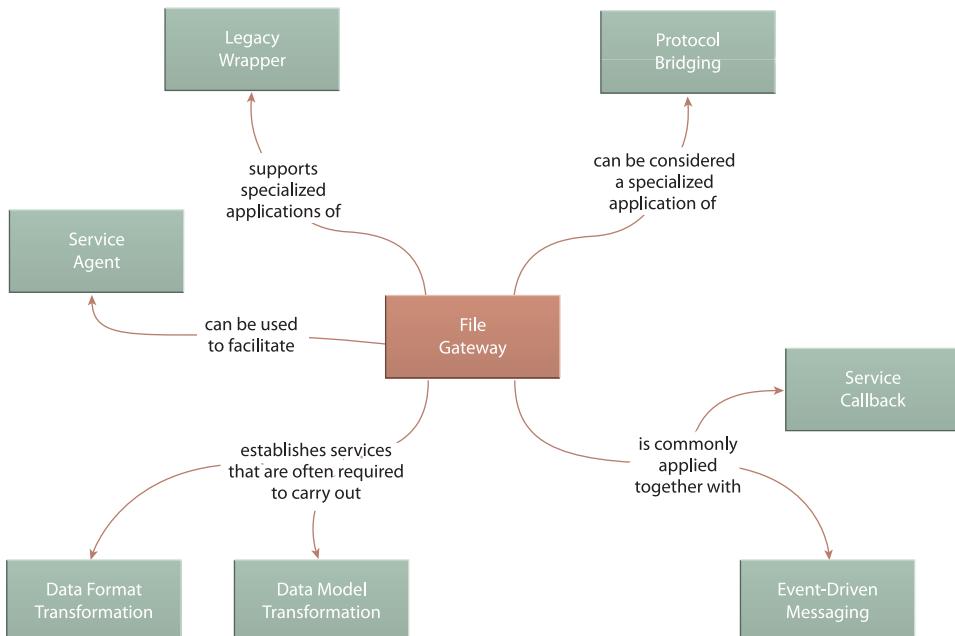


Figure 15.10

File Gateway establishes relationships with all of the transformation patterns associated with Service Broker (707).

File Gateway is also commonly applied together with Enterprise Service Bus (704) whereby the file gateway component is deployed within the bus platform and is designed to leverage its inherent broker and routing features.

NOTE

This pattern leverages and builds upon a number of established EAI patterns, such as File Transfer (Hohpe, Woolf).

CASE STUDY EXAMPLE

The FRC EDI system explained in the preceding example for Multi-Channel Endpoint (451) is currently subjected to a monthly batch extract of payment data that is manually imported into the accounting system. The batch extract produces a large set of flat files, with each file containing payment information for one transaction.

The FRC has also been developing a new Accounts Web service responsible for providing access to information pertaining to corporate accounts held by companies registered with the FRC. This service needs to be extended with a `GetPaymentHistory` operation that will retrieve payment records for a given account.

The payment details collected by the call center are immediately entered into the accounting database and are therefore readily accessible by the Accounts service. However, because the balance of the payment details that are stored in the EDI legacy repository are only imported monthly, the service will rarely gain access to current data.

Upon further investigation, architects discover an API provided by the legacy environment that allows extracts to be performed at any time. With this interface available to them, they decide to design an architecture with the following parts:

- a file gateway component capable of interacting with the API
- a WS-Addressing framework capable of supporting endpoint references

With this architecture, each time the `GetPaymentHistory` operation is called, the following processing steps occur:

1. The Accounts service invokes the file gateway component and passes it its data request along with its destination address, as per Service Callback (566).

2. The file gateway component initiates a file extract.
3. When the extraction is complete and requested data is available, the component contacts the Accounts service via the provided destination address.
4. The Accounts service is invoked and receives the data.

The internal file gateway logic is further optimized to poll the target directory for file drops in advance on a periodic basis so that the time required to initiate a batch extract for the most current data is reduced because a base of recent payment data has already been extracted.



Service Governance Patterns

Flexible Change

Identification

Notification

Refactoring

Decomposition

Capability

Proposed 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.

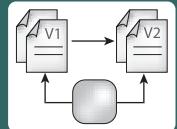
The most fundamental pattern in this chapter is Service Refactoring (484), which leverages a loosely (and ideally decoupled) contract to allow the underlying logic and implementation to be upgraded and improved.

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).

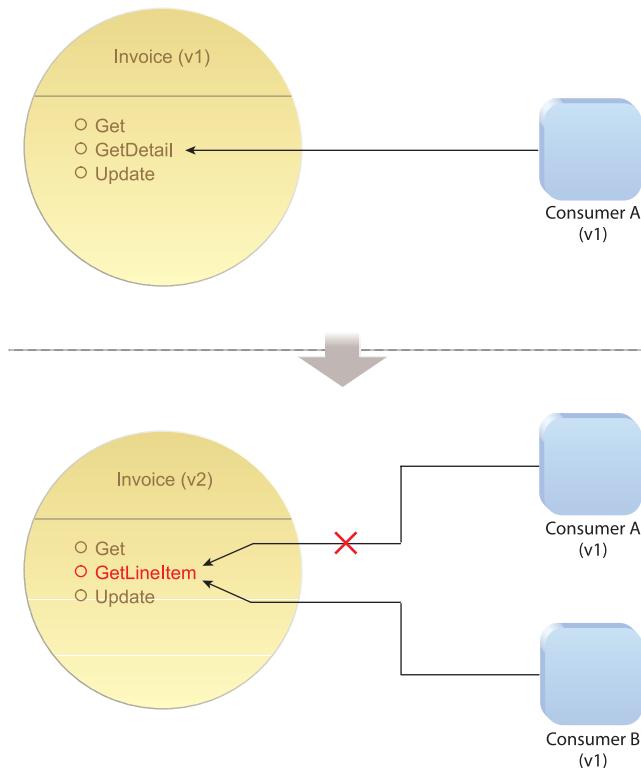


Figure 16.1

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

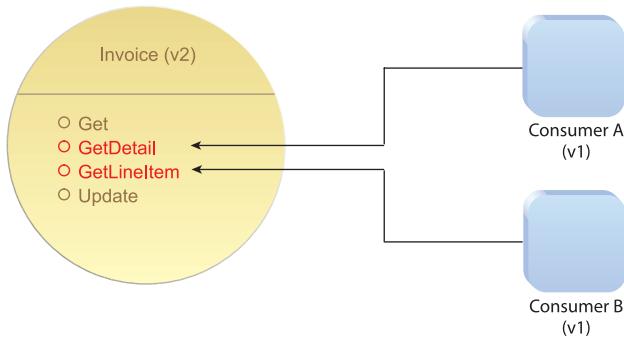


Figure 16.2

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.

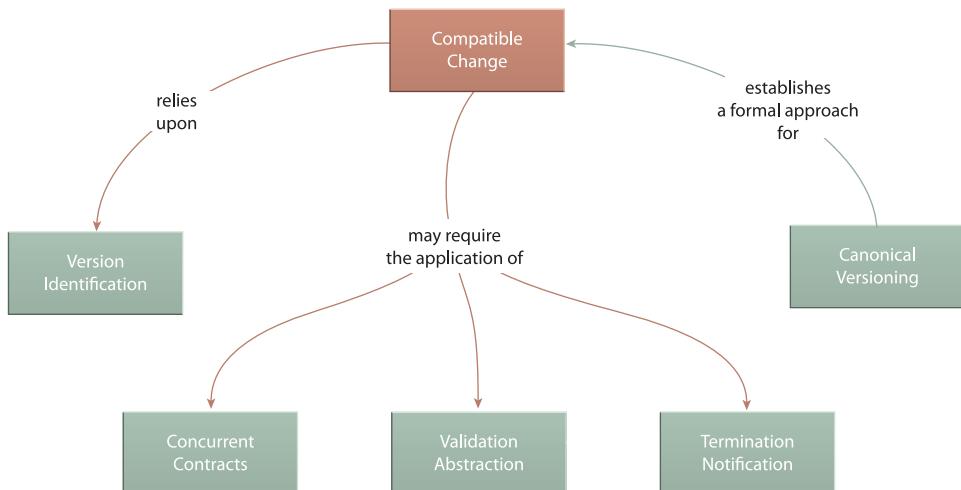


Figure 16.3

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>
```

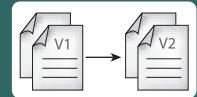
Example 16.1

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.

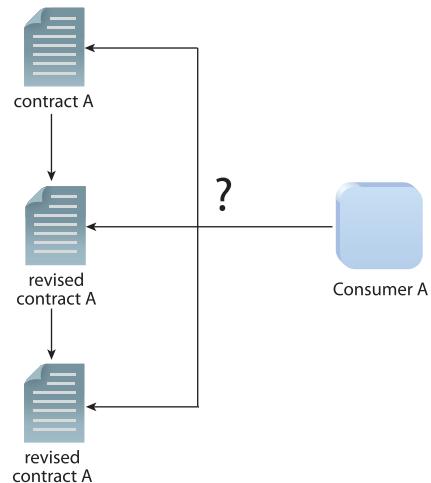


Figure 16.4

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.

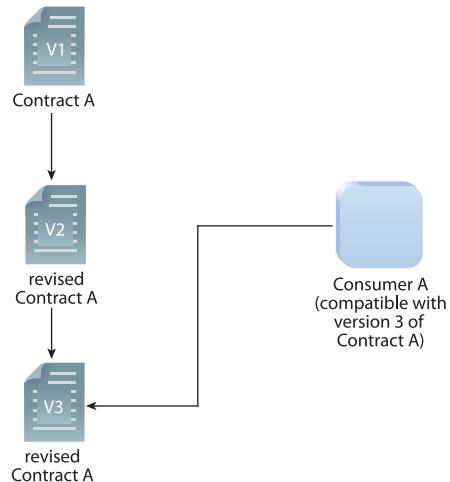


Figure 16.5

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).

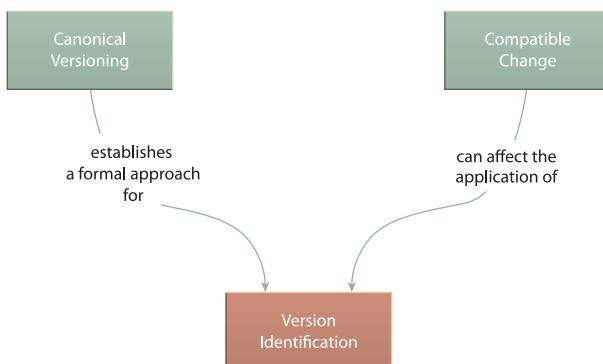


Figure 16.6

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/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">
  <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>
```

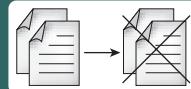
```
</portType>
...
</definitions>
```

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).

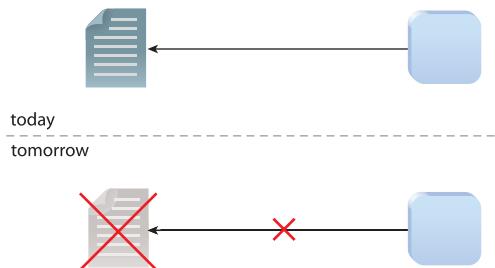


Figure 16.7

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).

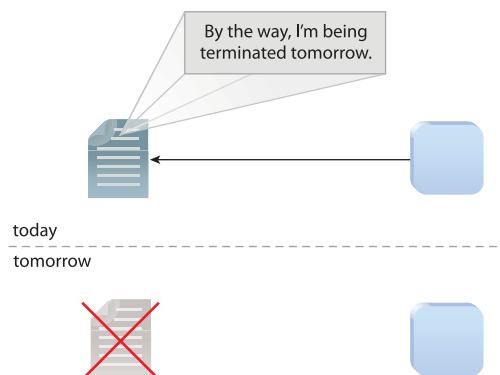


Figure 16.8

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.

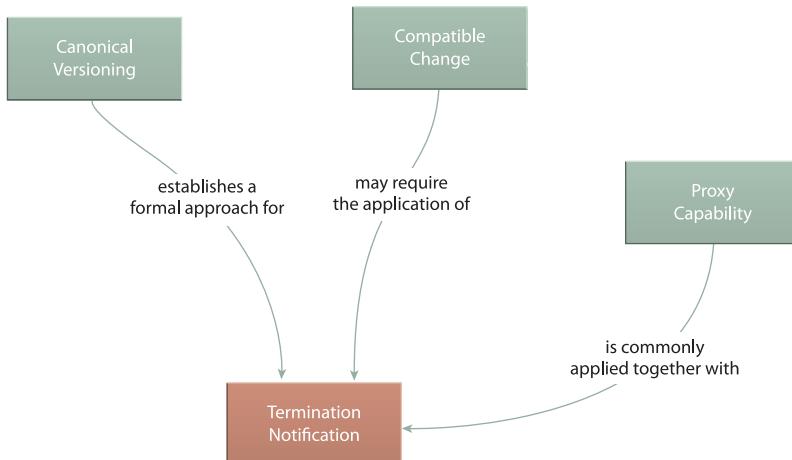


Figure 16.9

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>
```

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

Example 16.4

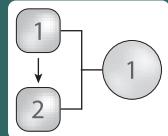
The `operation` element within the Officer WSDL definition's `binding` construct is modified to include a custom ignorable WS-Policy assertion that expresses the scheduled termination date of the `UpdateLog` operation.

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?



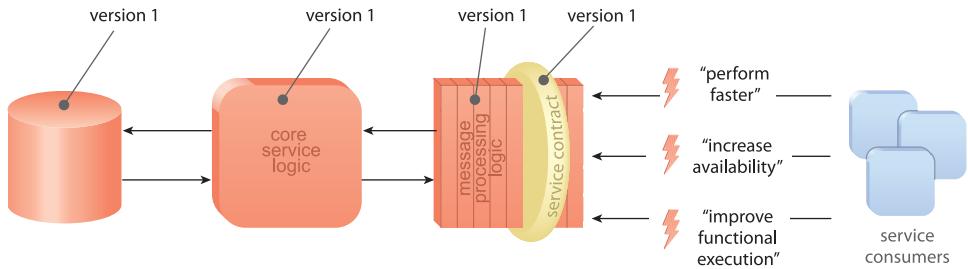
Problem	The logic or implementation technology of a service may become outdated or inadequate over time, but the service has become too entrenched to be replaced.
Solution	The service contract is preserved to maintain existing consumer dependencies, but the underlying service logic and/or implementation 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

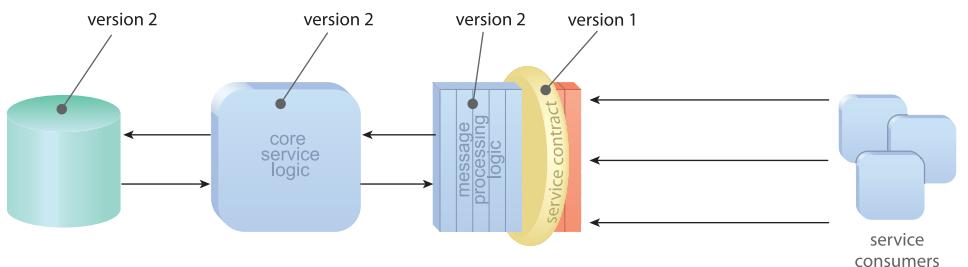
Subsequent to its initial delivery, unforeseen performance and business requirements may demand more from a service than it is capable of providing (Figure 16.10). Replacing the service entirely may be undesirable, especially when several consumer programs have already formed dependencies upon its established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

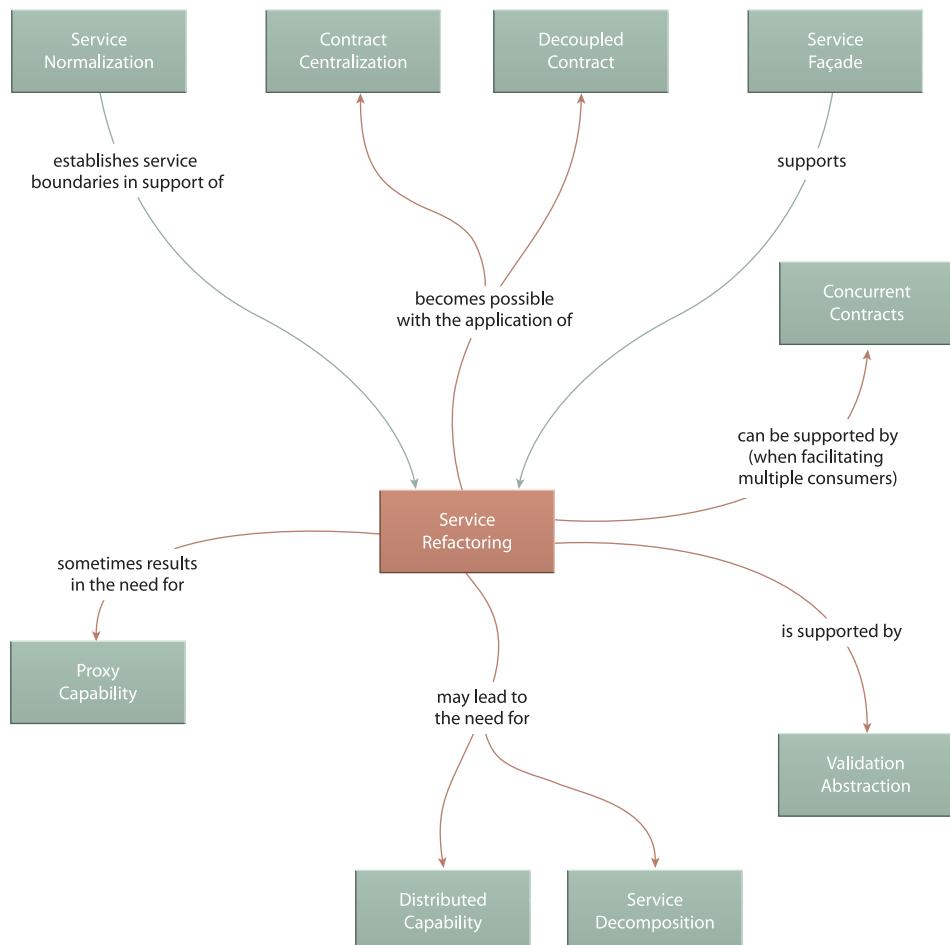


Figure 16.12

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.

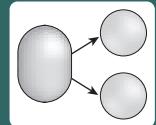
As a result, it was decided to completely replace this product. This, of course, affected many services, including the Employee service. However, because its contract was decoupled and had been fully standardized, it was in no way dependent on any part of the underlying ERP environment.

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

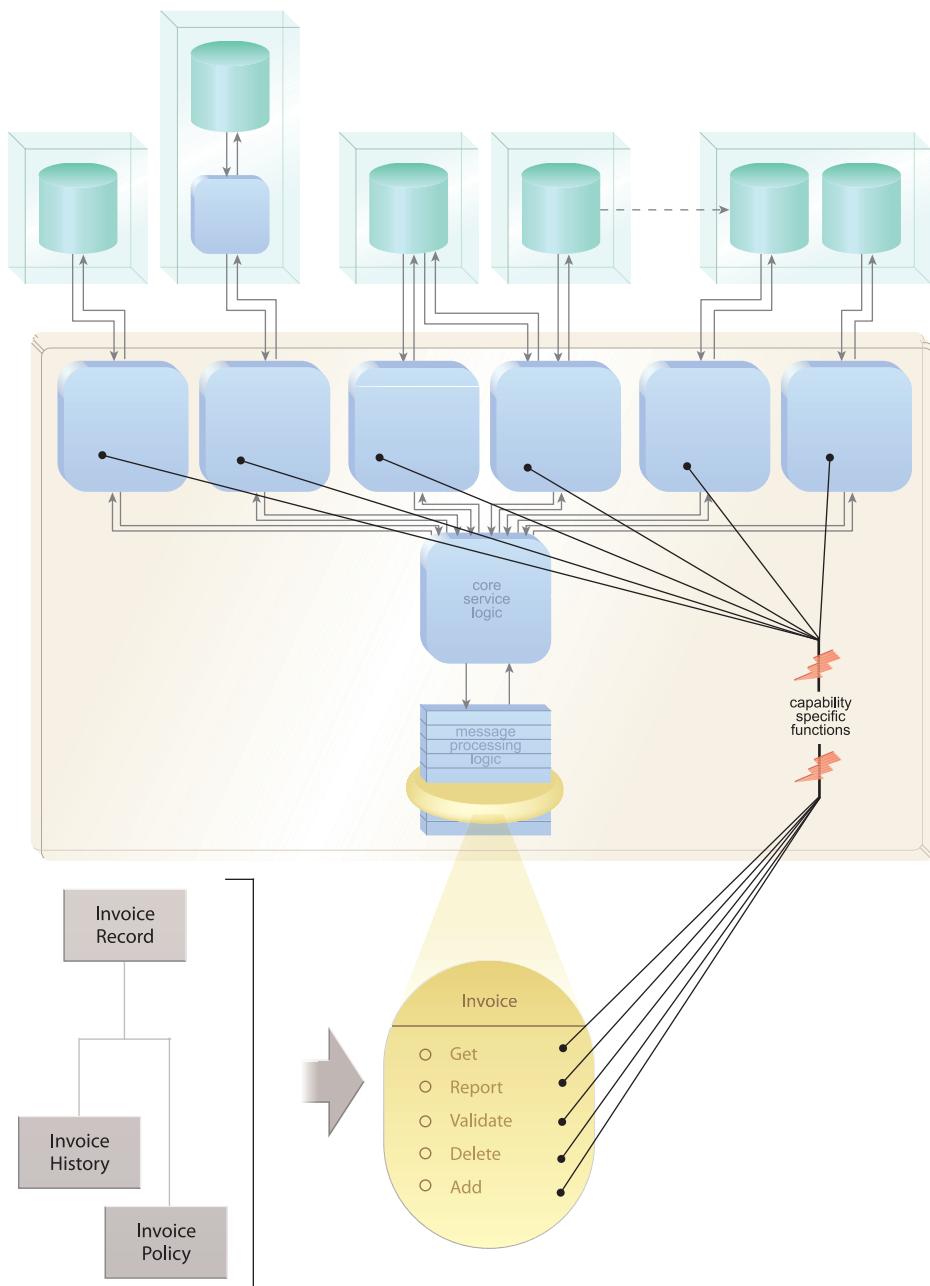
Table 16.5

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).

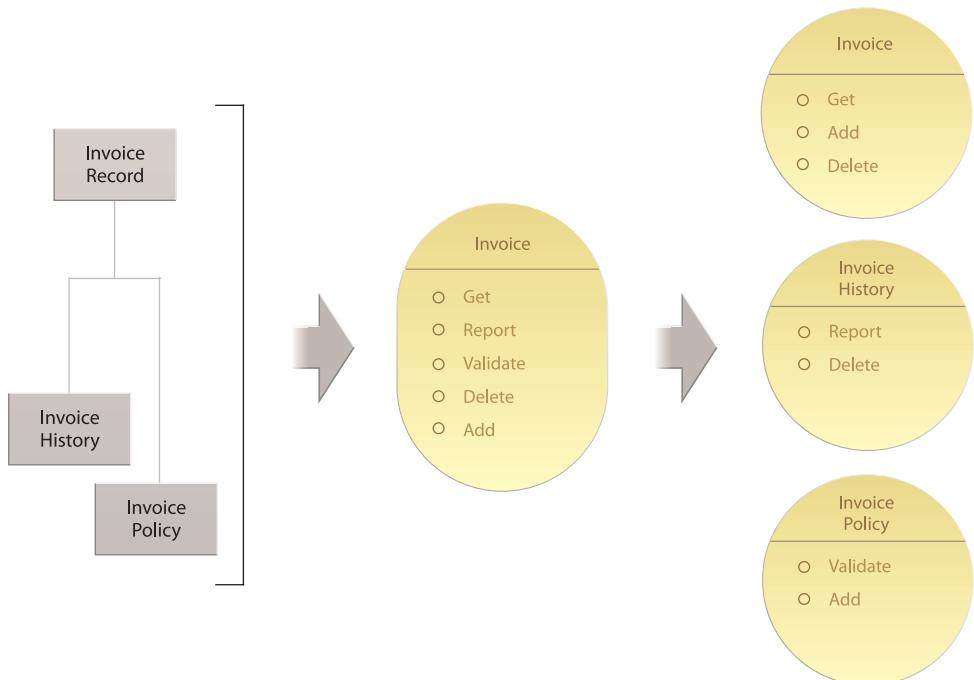


Figure 16.14

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

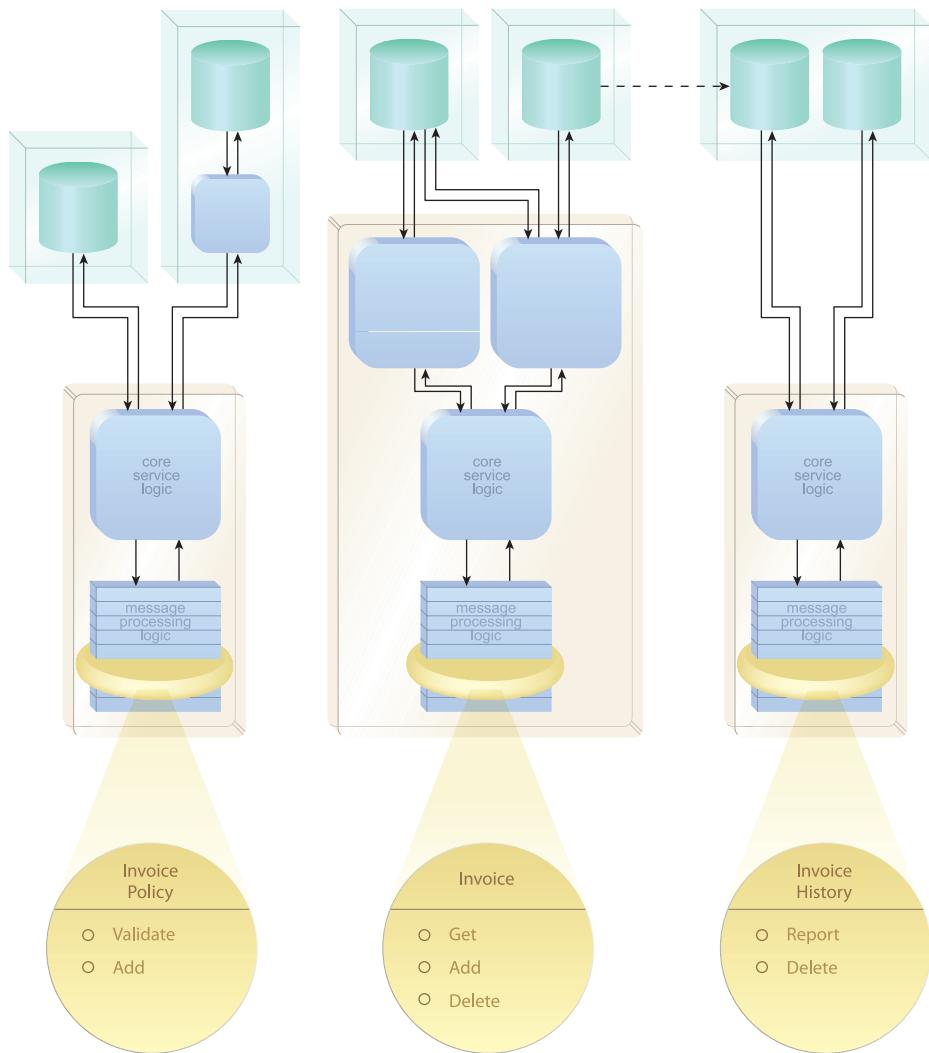
The concepts behind this pattern can also be applied in reverse, where two or more fine-grained services are combined into one coarse-grained service. The use of Proxy Capability (497) would still apply for preserving the original service contracts. This is the basis of a pattern called Service Consolidation 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.

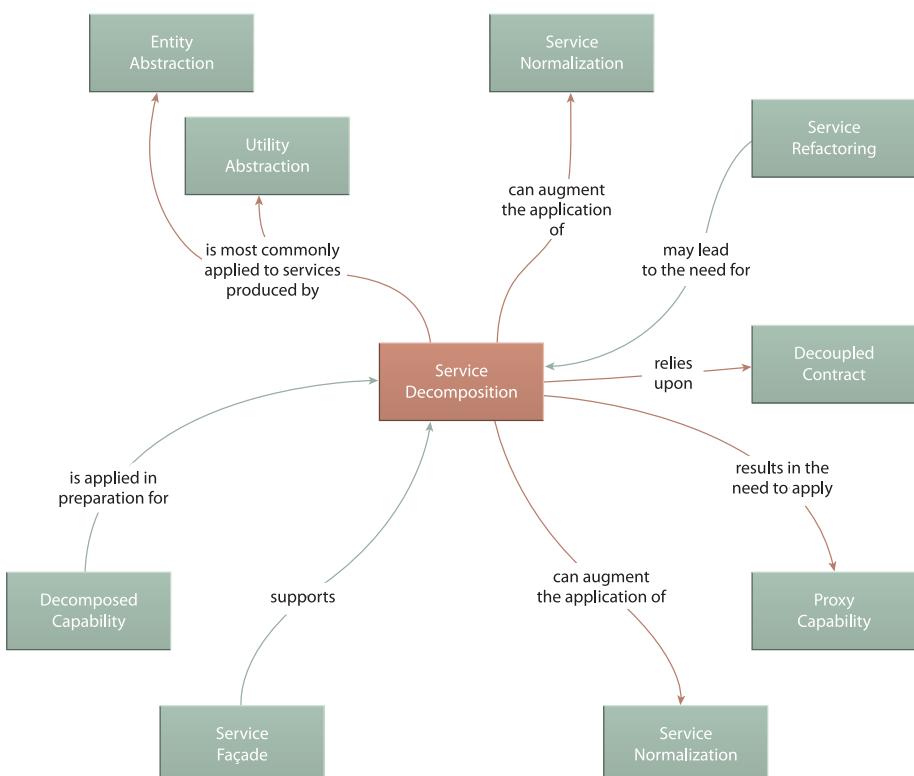


Figure 16.16

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.

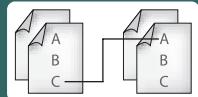
After taking all these factors into consideration, the team feels that it makes sense to break off historical reporting functionality into a separate service appropriately called “Employee Records.” The first challenge they face is that the existing Employee service contract is already being used by 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).

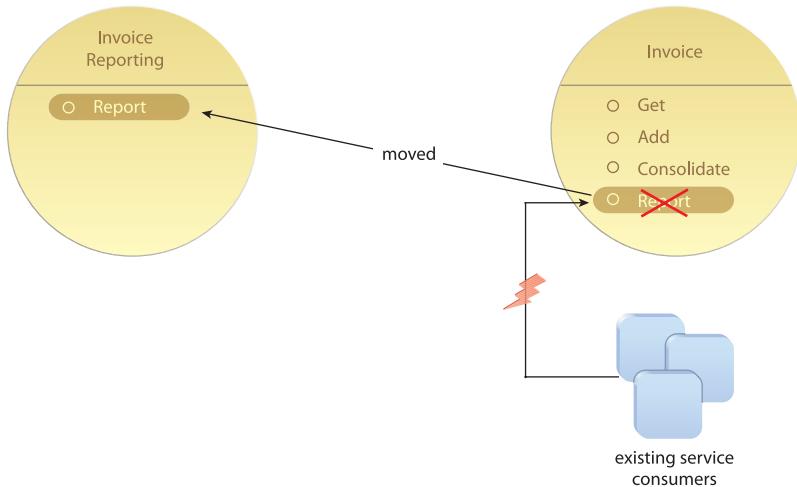


Figure 16.17

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

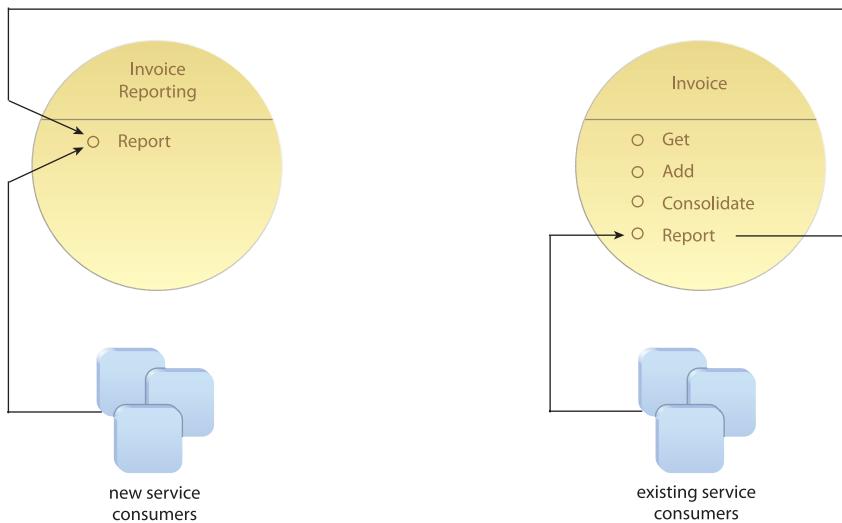
Solution

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).

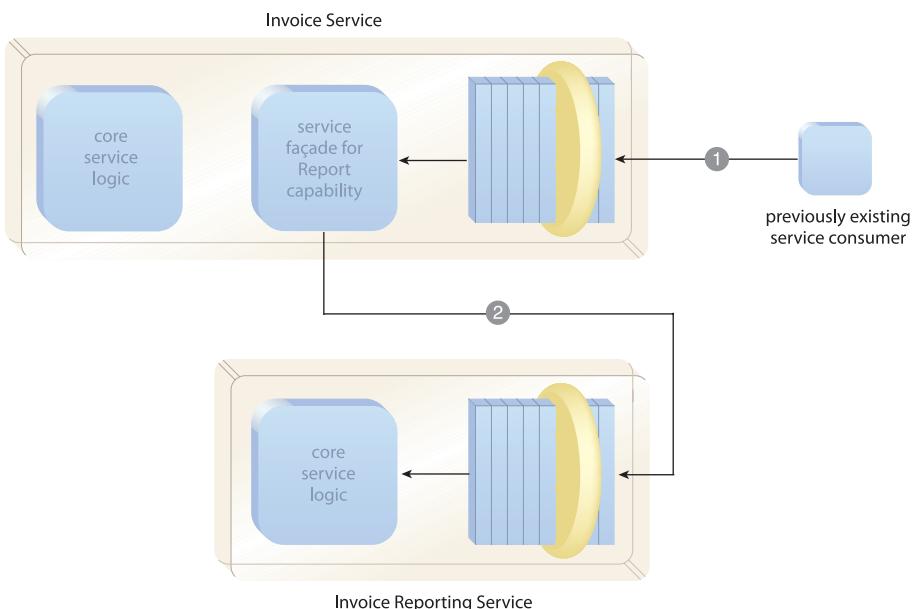
This does not prevent the capabilities in the new services from being independently accessed. In fact, access to the capability logic via its new service contract is encouraged so as to minimize the eventual effort for proxy capabilities to be phased out.

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.

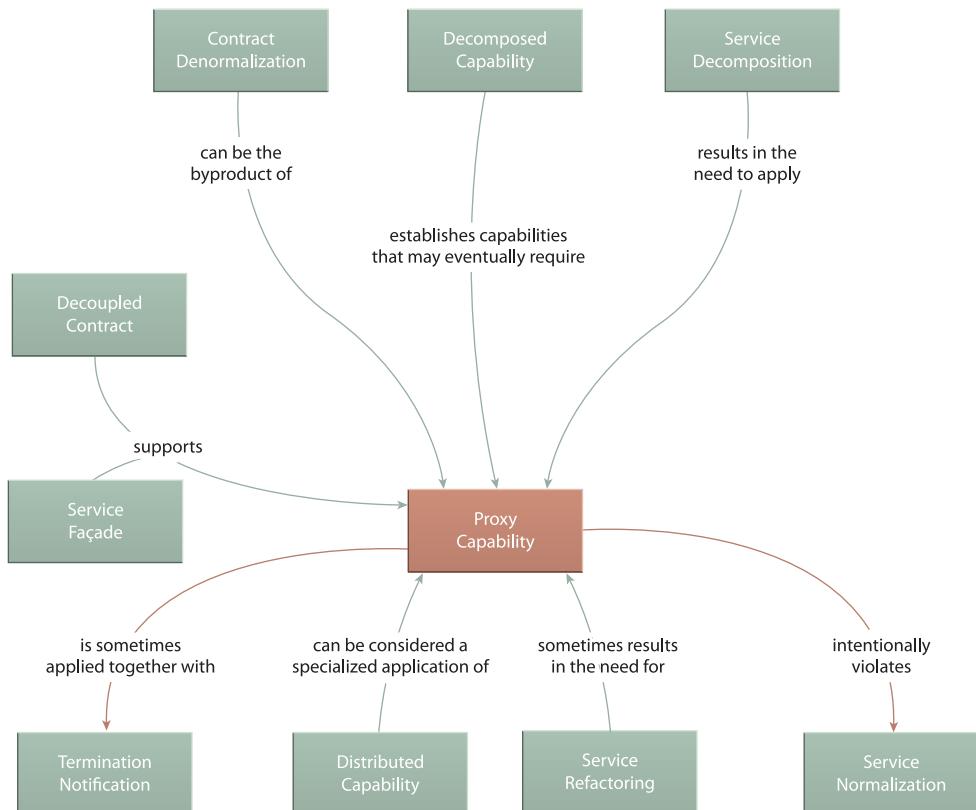
Furthermore, this pattern alone does not guarantee that a proxy capability will continue to provide the same behavior and reliability of the original capability it replaced.

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.

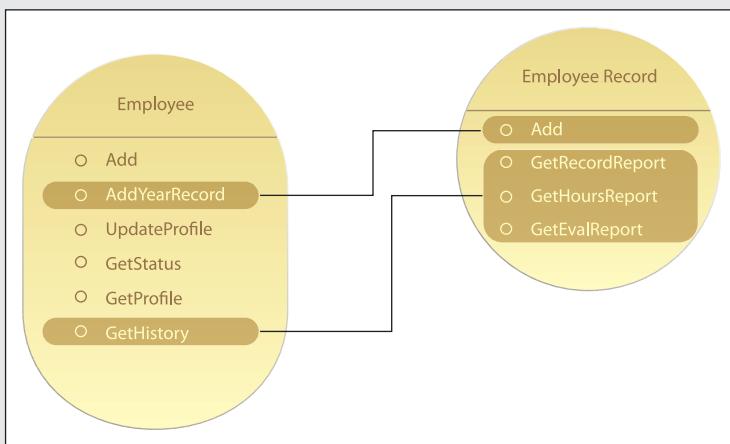


Figure 16.22

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

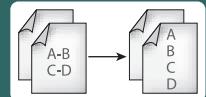
The Employee Record service is eventually designed and delivered as a fully functional, standalone service. However, the Employee service contract remains unchanged, plus additional logic is added in the form of a façade component. This functionality responds to requests for the original AddRecord and 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.

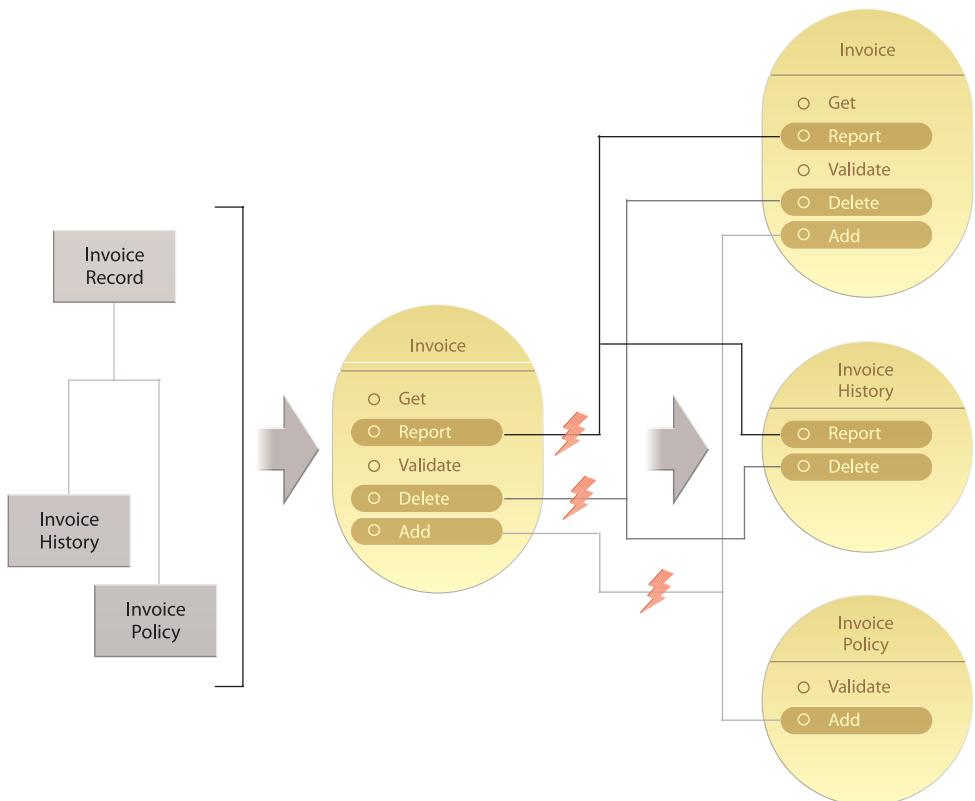


Figure 16.23

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).

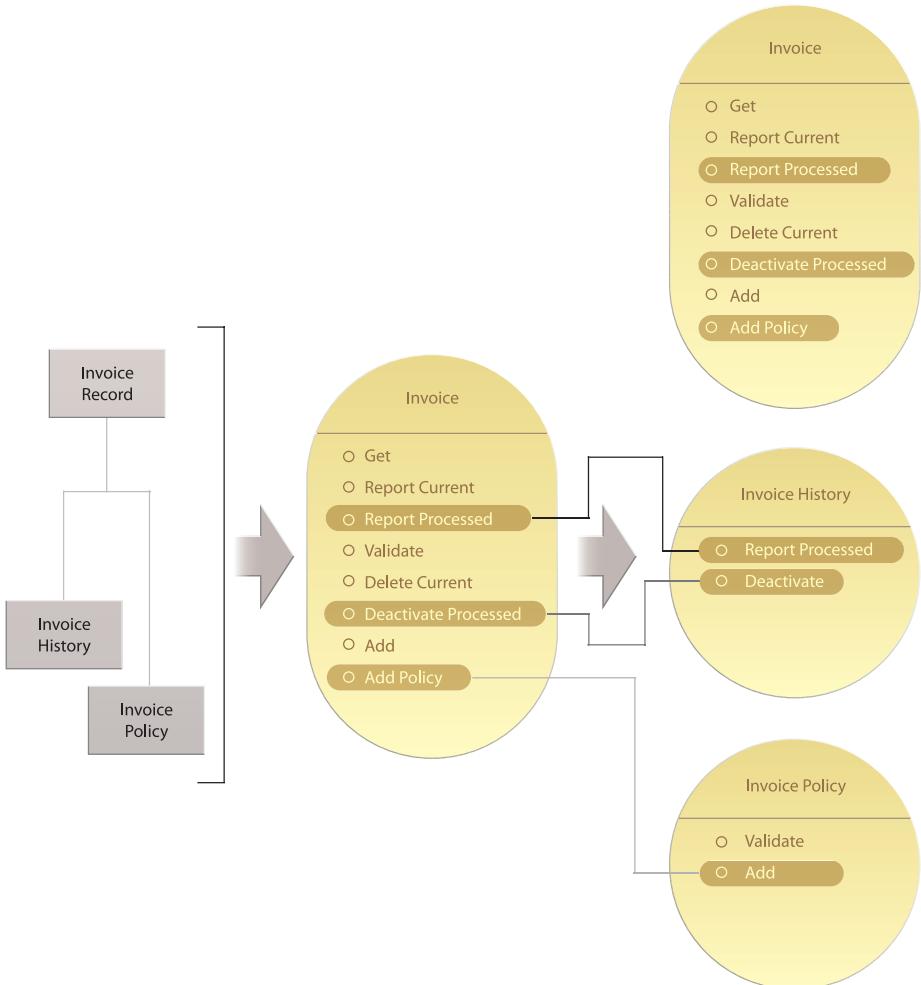


Figure 16.24

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.

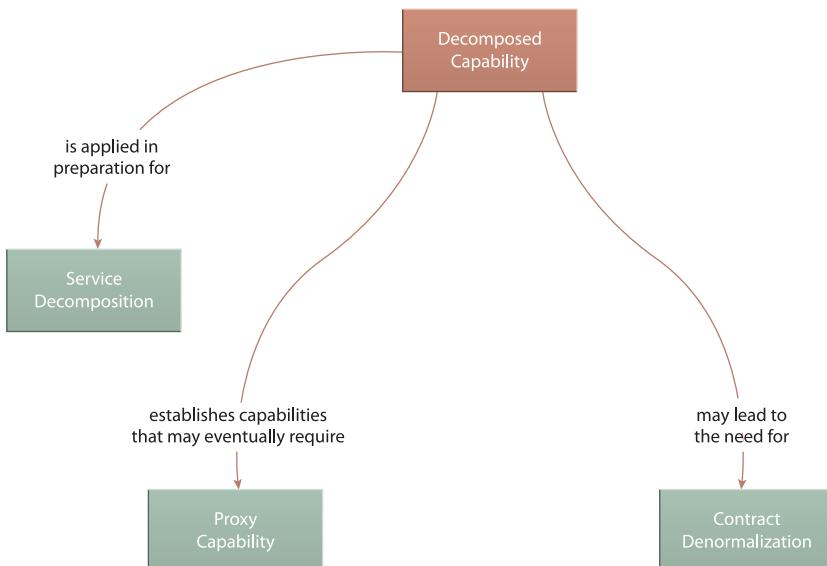


Figure 16.25

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.

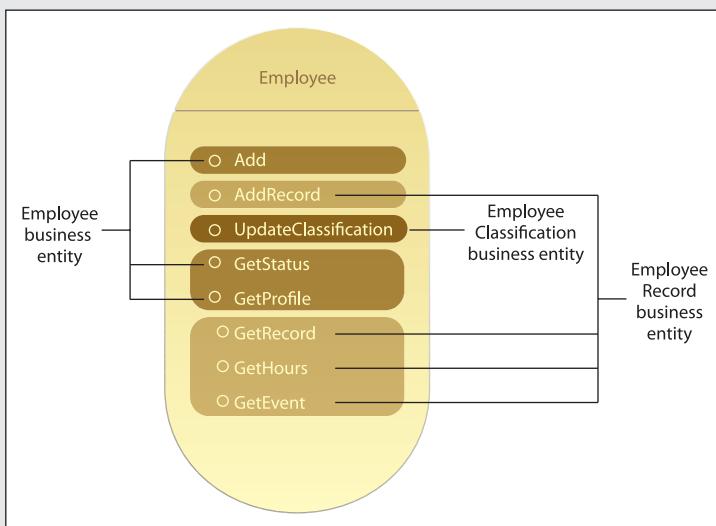
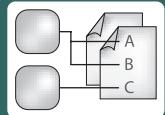


Figure 16.26

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).

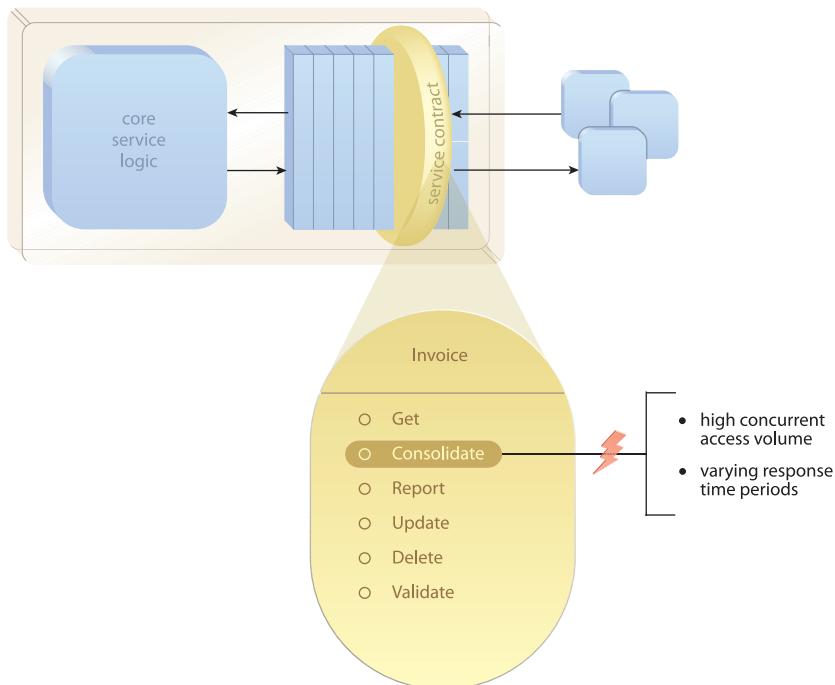


Figure 16.27

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).

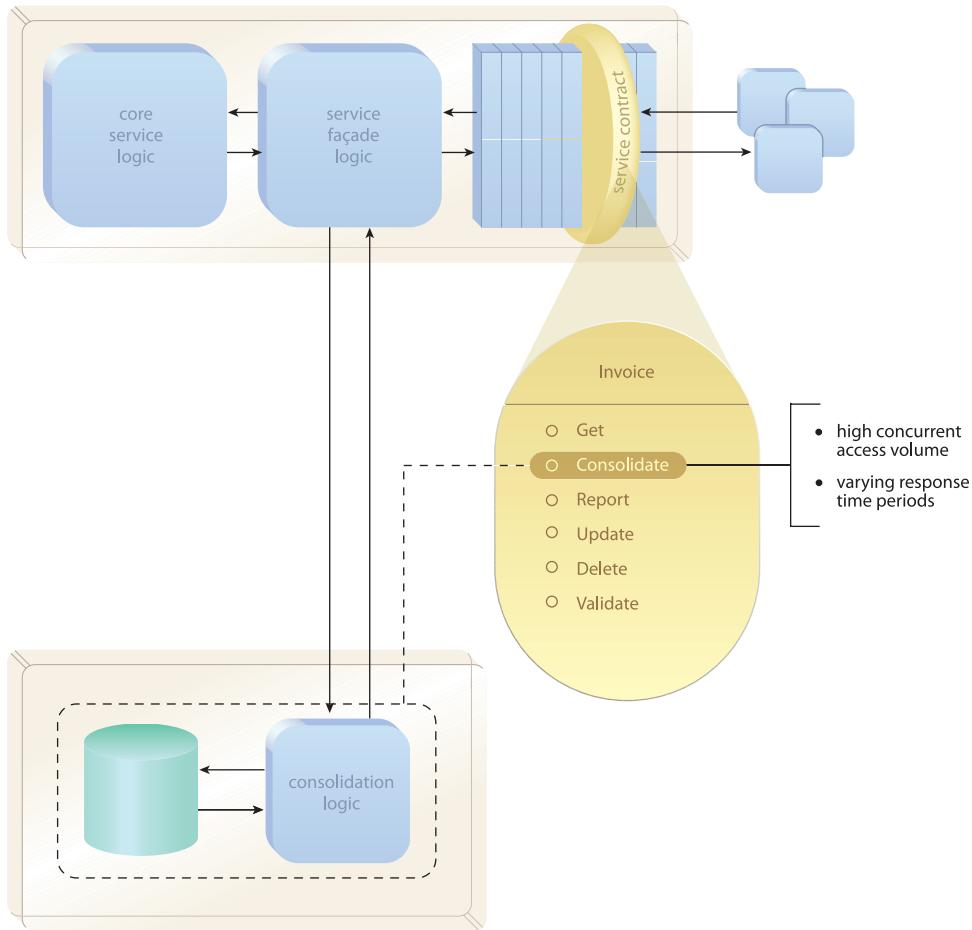


Figure 16.28

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.

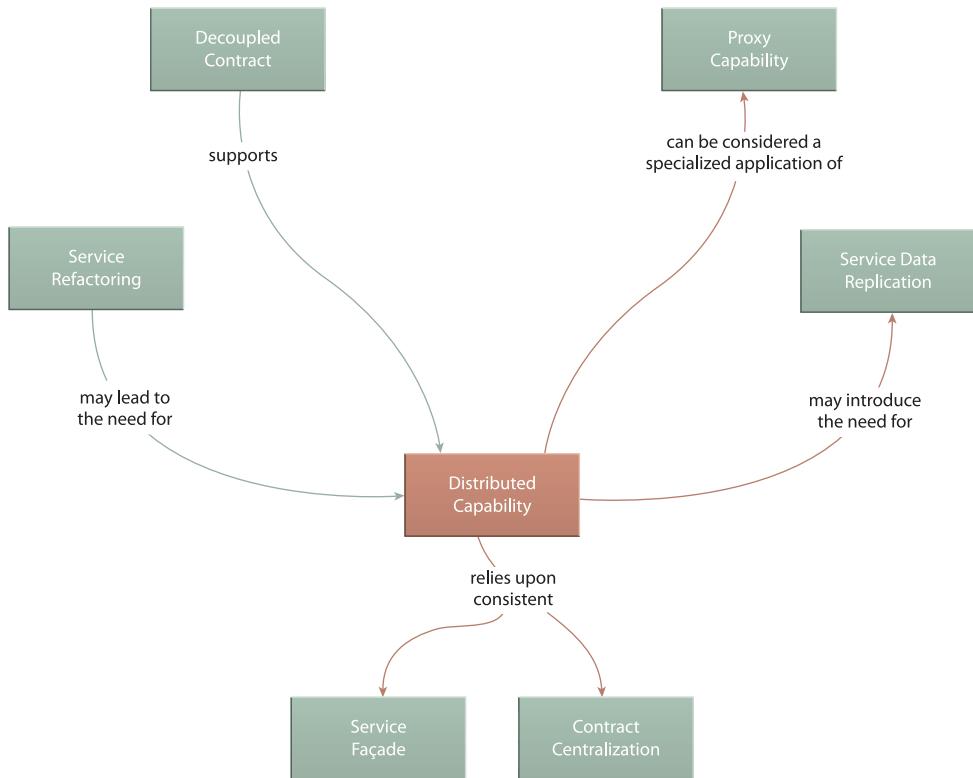
If the capability logic was separated to guarantee a certain response time during high volume usage, then this may be somewhat undermined by the remote access requirements. On the other hand, overall service autonomy tends to be positively impacted as the autonomy level of the separated capability logic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

Part IV



Service Composition Design Patterns

Chapter 17: Capability Composition Patterns

Chapter 18: Service Messaging Patterns

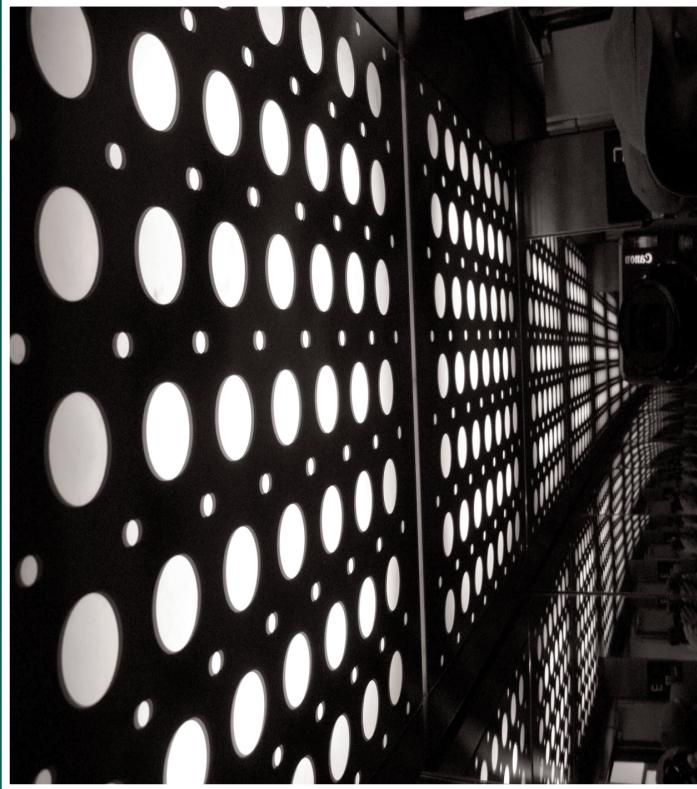
Chapter 19: Composition Implementation Patterns

Chapter 20: Service Interaction Security Patterns

Chapter 21: Transformation Patterns

This page intentionally left blank

Chapter 17



Capability Composition Patterns

Capability Composition

Capability Recomposition

The patterns in this chapter provide the means by which to assemble and compose together the service logic that was successfully decomposed, partitioned, and streamlined via the previously explained service identification and definition patterns (Figure 17.1).

These composition patterns essentially establish capabilities as the fundamental means by which service logic is aggregated to solve one or more larger problems. Studying these patterns reveals how composition logic becomes a natural part of service design.

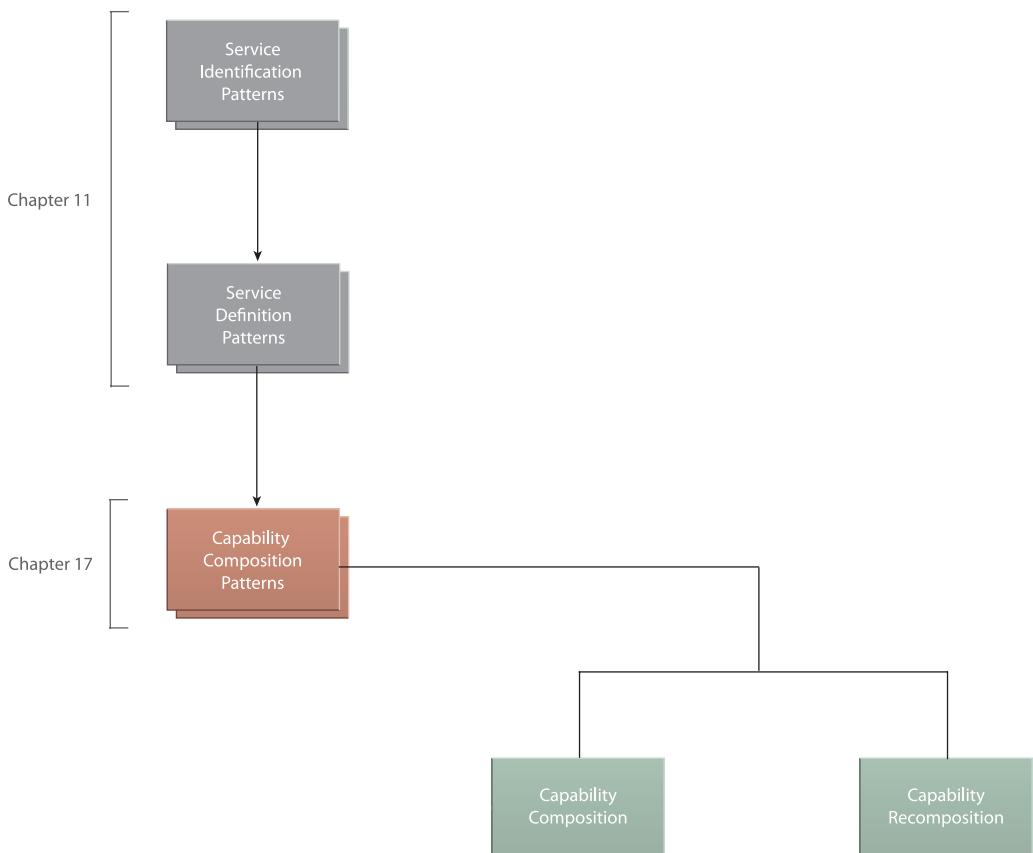
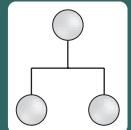


Figure 17.1

Capability composition patterns build upon service identification and definition patterns to establish the concept of service composition.

Capability Composition

How can a service capability solve a problem that requires logic outside of the service boundary?



Problem	A capability may not be able to fulfill its processing requirements without adding logic that resides outside of its service's functional context, thereby compromising the integrity of the service context and risking service denormalization.
Solution	When requiring access to logic that falls outside of a service's boundary, capability logic within the service is designed to compose one or more capabilities in other services.
Application	The functionality encapsulated by a capability includes logic that can invoke other capabilities from other services.
Impacts	Carrying out composition logic requires external invocation, which adds performance overhead and decreases service autonomy.
Principles	All
Architecture	Inventory, Composition, Service

Table 17.1

Profile summary for the Capability Composition pattern.

Problem

Although the nature of a capability may be in alignment with a service's overall functional context, the logic required to carry out the capability may need to go beyond the designated service context boundary.

The service boundary could be increased, but this would change its original context and further introduces the danger of functional overlap and service denormalization because the expanded boundary could infringe on the functional boundaries of other services.

Solution

A service capability does not execute logic that resides outside of the service's functional boundary. Instead, it invokes the appropriate capability in a different service based on the appropriate functional boundary (Figure 17.2).

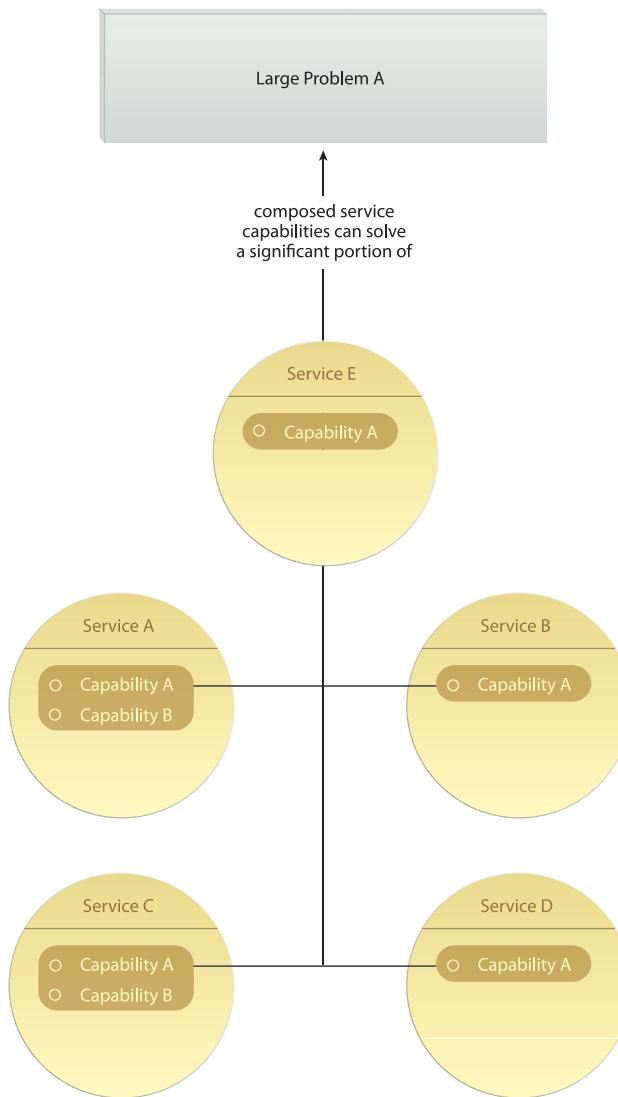


Figure 17.2

The individual capabilities of services can be aggregated to collectively help solve the large problem from which they were originally derived.

Application

This pattern is applied throughout a service delivery lifecycle. For example:

- During the service modeling phase, composition candidates are assembled to define conceptual aggregates comprised of individually composed capability candidates.
- The service design process requires that the functional processing requirements of a service capability be analyzed so as to identify the potential involvement of capabilities.
- When in development, distributed invocation logic may need to be embedded within the capability routines, especially when required to access capabilities residing in other physical services.

Note that if an external body of logic is defined for which no service capability yet exists, then that logic needs to be created as part of a new capability (not as part of the existing capability). The new capability may or may not form the basis of a new service.

Impacts

When capabilities are distributed across numerous services, some of which may reside in remote locations, cross-service capability invocation can impose measurable runtime performance overhead.

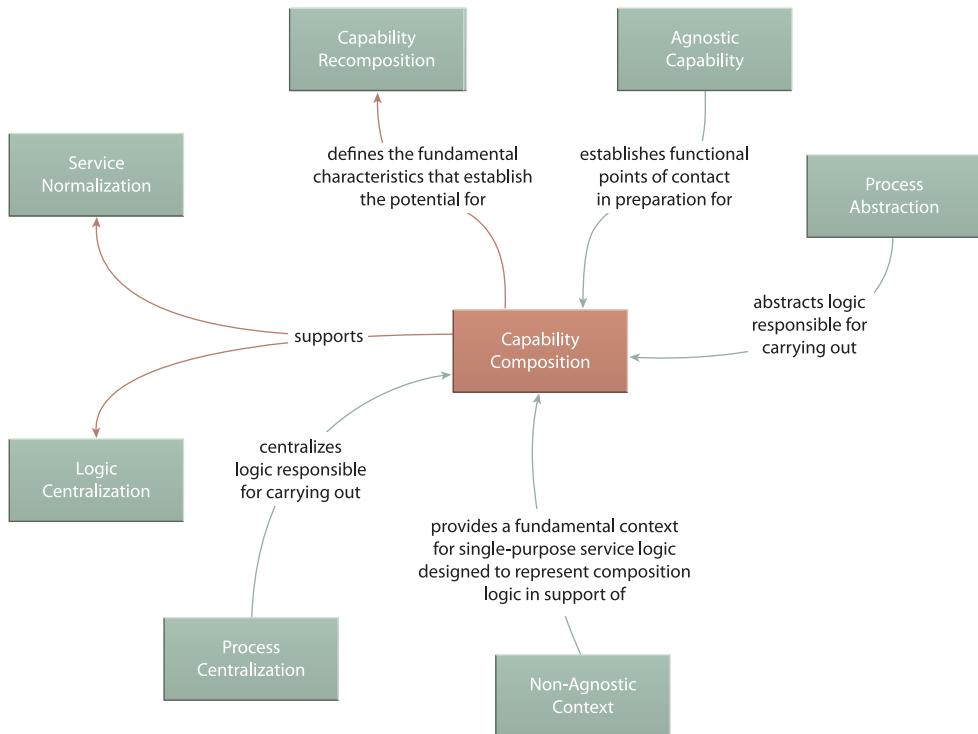
Also the overall autonomy of a service is reduced due to the fact that its capability is dependent on another service. This eventuality represents an important design dynamic within service compositions that the application of the Service Autonomy design principle helps prepare a service for.

Furthermore, requiring that a new capability be created when the required external logic does not exist can lead to unexpected scope increases in service delivery projects.

Relationships

Because service-oriented computing is a distributed computing platform, it is fully expected that a solution is comprised of parts that are aggregated together. However, Capability Composition does more than just enable service aggregation. It ensures that Service Normalization (131) and Logic Centralization (136) are fully supported by requiring external service invocation through service boundary enforcement.

It is further important that this pattern be viewed as a step toward what services and their supporting architectures must ultimately realize: Capability Recomposition (526).

**Figure 17.3**

Capability Composition represents the ability for services to be composed but not necessarily repeatedly.

NOTE

The following case study example is a continuation of the examples in Chapter 11.

CASE STUDY EXAMPLE

The segment of the Cutit Chain Inventory Transfer process that requires the creation of an inventory record and the cross-referencing of associated back orders could represent a body of logic that is always required with each newly created inventory record. Therefore, this query could be automatically carried out as part of the Inventory service's Add capability.

However, this pattern dictates that the Add capability only create an inventory record and nothing more. Issuing data queries against order information is outside of the capability's functional boundary and also beyond the parent service scope of inventory-related processing.

The ability for a back order query to be issued is already present within the Order service's GetBackOrders capability. Therefore, if the Inventory service's Add capability requires this functionality, it simply needs to invoke that capability via the Order service. Although this imposes performance requirements associated with multiple service invocation, it guarantees that no one service will functionally overlap with others.

The application of this pattern on a broader scale results in the assembly of all four services into a coordinated composition capable of automating the Chain Inventory Transfer business process, as shown in Figure 17.4.

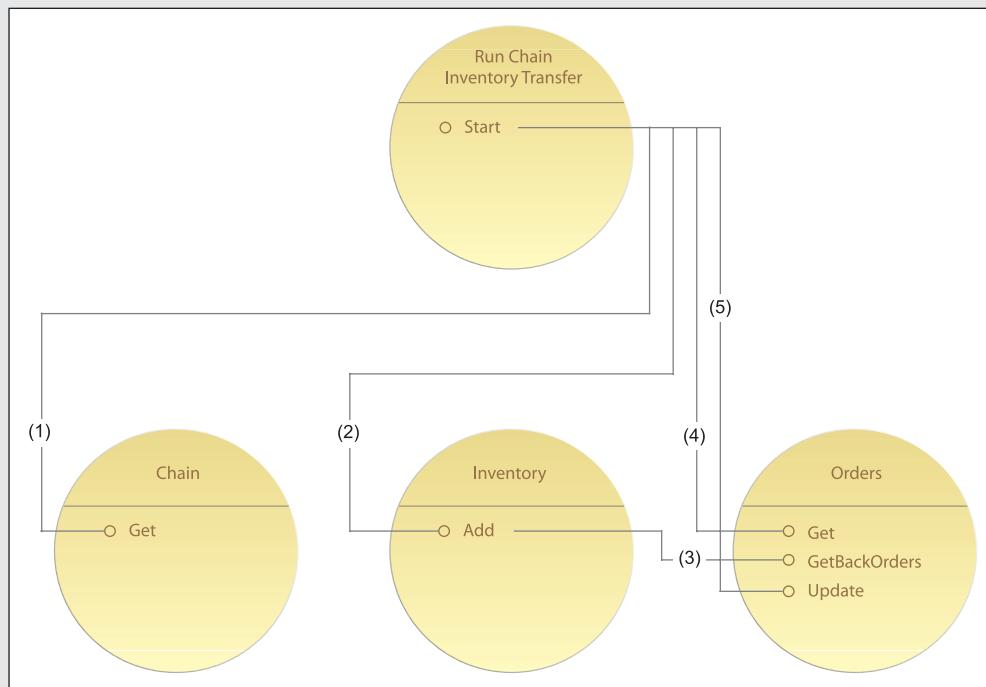
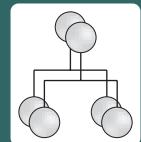


Figure 17.4

The final composition of the services defined via the application of this pattern to all capabilities in support of the coordinated automation of the Chain Inventory Transfer process. Note composition step 3 representing the newly introduced feature of the Inventory service's Add capability to perform automatic back order checks.

Capability Recomposition

How can the same capability be used to help solve multiple problems?



Problem	Using agnostic service logic to only solve a single problem is wasteful and does not leverage the logic's reuse potential.
Solution	Agnostic service capabilities can be designed to be repeatedly invoked in support of multiple compositions that solve multiple problems.
Application	Effective recomposition requires the coordinated, successful, and repeated application of several additional patterns.
Impacts	Repeated service composition demands existing and persistent standardization and governance.
Principles	All
Architecture	Inventory, Composition, Service

Table 17.2

Profile summary for the Capability Recomposition pattern.

Problem

A distributed solution can be comprised of services designed for a specific composition. This is often the case when collections of services are delivered by independent projects. Because these services are tuned to automate a particular business process, little consideration is given to their potential to solve other business problems.

As a result, the outstanding business problems get solved with new collections of services. This approach enables individual solutions to distribute logic in response to immediate concerns (such as special security and performance requirements) but does not foster reuse to any meaningful extent.

Typical effects associated with missing reuse opportunities arise, reminiscent of silo-based environments—for example, the proliferation of redundant logic, wasteful delivery projects, and an increasingly bloated enterprise.

Solution

A key fundamental pattern and one that is essential to the realization of most strategic service-oriented computing goals is that of Capability Recomposition. The successful application of this pattern allows agnostic logic to be repeatedly reused as part of different service aggregates assembled to solve different problems (Figure 17.5).

Application

Though fundamental, this is a pattern very much dependent on others in that it builds upon and leverages many of the patterns in this book. In fact, the extent to which Capability Recomposition can be realized is dependent on the extent to which other SOA patterns have been and will continue to be successfully applied.

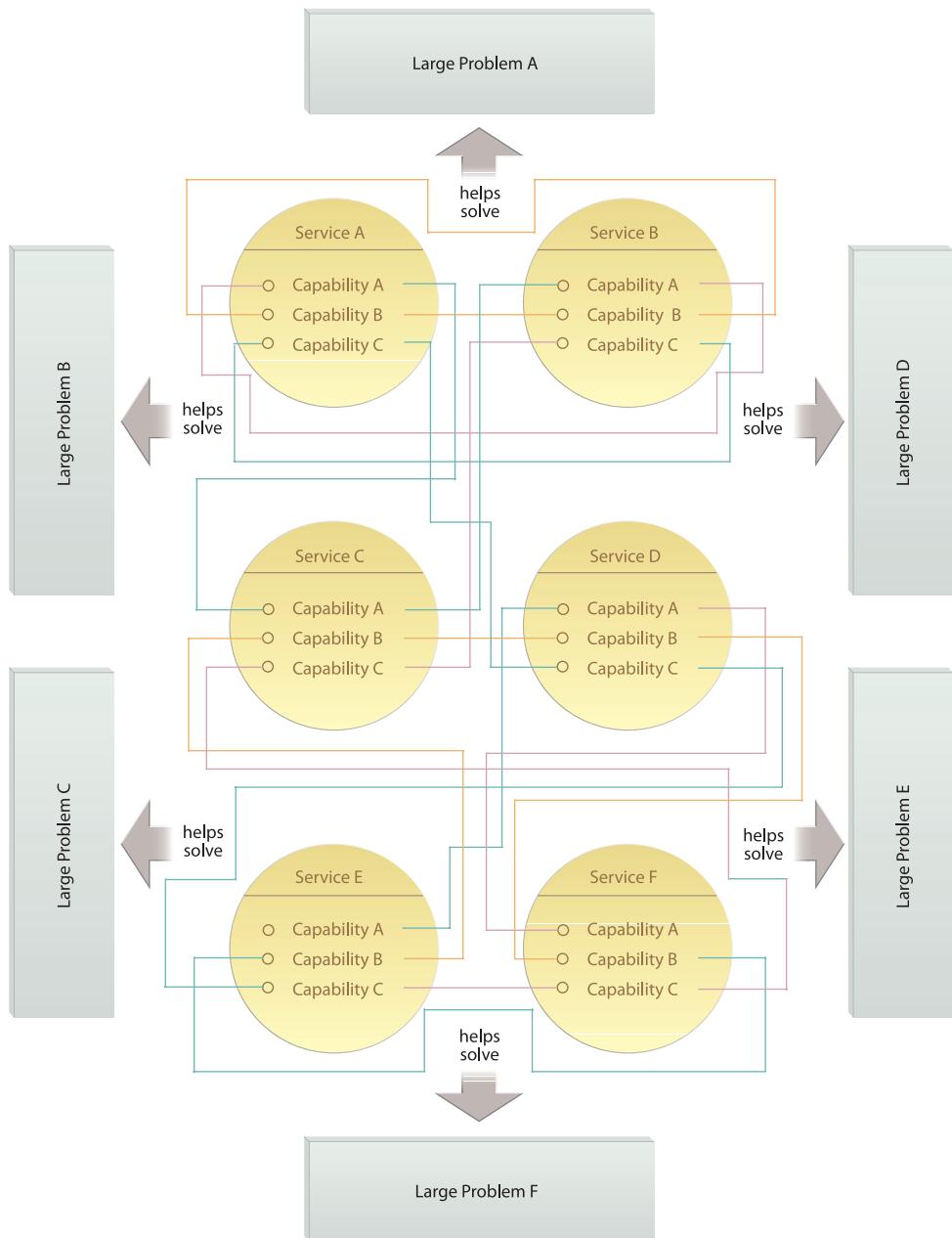
So what's the difference between this design pattern and the repeated application of Capability Composition (S21)? For the logic encapsulated by a capability to be repeatedly composable, it must be designed in such a manner that it can facilitate numerous scenarios and concurrent invocation. These are not requirements for realizing Capability Composition (S21).

It is worth noting that the Service Composability design principle is dedicated to supporting the goals of this pattern. The design considerations raised by this principle help ensure that other service-orientation principles are sufficiently applied so that each service capability is prepared for recomposition.

Impacts

Just as this pattern results in strategic benefits from the combined application of other patterns, it also inherits their collective challenges and complexities. Service composition itself represents a design technique that may impose a learning curve upon those responsible for solution design. It is comprised of a unique process that requires a combination of creativity and awareness of how services can be effectively combined within the constraints of the underlying architecture and infrastructure.

Furthermore, the importance of governing agnostic services is greatly amplified, as these represent the parts of a service inventory most prone to repeated composition. Performance, security, version control, and interaction requirements of each agnostic service can impact the design of any given service composition. Service ownership also plays a key role in ensuring that agnostic services are properly evolved throughout participation in multiple compositions.

**Figure 17.5**

The individual capabilities of the original services can be repeatedly aggregated together with additional capabilities into different composition configurations. This enables capabilities to collectively solve the large problem for which they were originally delivered in addition to several other problems.

Relationships

Many patterns in this book support Capability Recomposition. This multitude of supportive relationships is key to understanding the dynamics behind service-orientation. Ultimately, the repeated composition of service capabilities is what leads to the attainment of several of the key strategic benefits and goals associated with service-oriented computing.

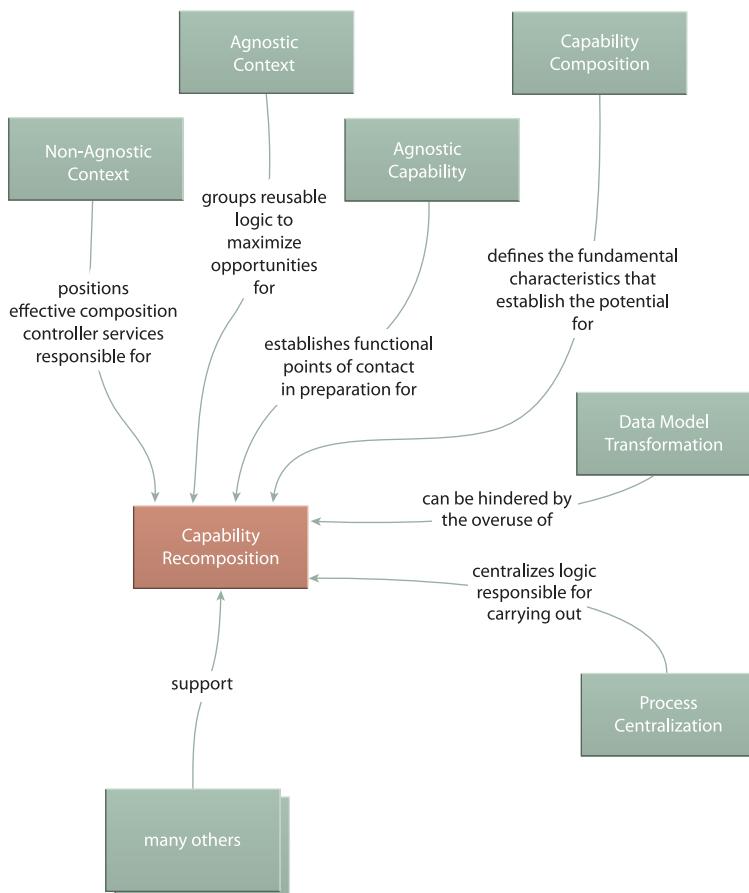


Figure 17.6

Numerous patterns relate to and support Capability Recomposition.

CASE STUDY EXAMPLE

All of the services created for the automation of the Chain Inventory Transfer process offer logic that is reusable and expressed via standardized service contracts. This makes the logic well-suited for recomposition for the purpose of automating other business processes and solving other problems.

Chapter 18



Service Messaging Patterns

Service Messaging

Messaging Metadata

Service Agent

Intermediate Routing

State Messaging

Service Callback

Service Instance Routing

Asynchronous Queuing

Reliable Messaging

Event-Driven Messaging

Numerous factors can come into play when designing the possible runtime activity that can occur between services within a composition. These patterns provide various techniques for processing and coordinating data exchanges between services.

Service Messaging (533) establishes the base pattern that others in this chapter further specialize and build upon. Messaging Metadata (538), for example, extends Service Messaging (533) by providing the opportunity to supplement messages with additional meta details. Transparent intermediary processing is provided by Service Agent (543) as well as the more specialized Intermediate Routing (549).

The Service Instance Routing (574), Service Callback (566), and State Messaging (557) patterns explore creative ways to leverage a messaging framework in order to communicate between service instances, form asynchronous messaging interactions, and defer state data to the message layer, respectively.

Finally, Asynchronous Queuing (582) and Reliable Messaging (592) provide inventory-level extensions that can improve the quality and integrity of message-based communication, and Event-Driven Messaging (599) establishes the well-known publish-and-subscribe messaging model in support of service interaction.

Service Messaging

How can services interoperate without forming persistent, tightly coupled connections?



Problem	Services that depend on traditional remote communication protocols impose the need for persistent connections and tightly coupled data exchanges, increasing consumer dependencies and limiting service reuse potential.
Solution	Services can be designed to interact via a messaging-based technology, which removes the need for persistent connections and reduces coupling requirements.
Application	A messaging framework needs to be established, and services need to be designed to use it.
Impacts	Messaging technology brings with it QoS concerns such as reliable delivery, security, performance, and transactions.
Principles	Standardized Service Contract, Service Loose Coupling
Architecture	Inventory, Composition, Service

Table 18.1

Profile summary for the Service Messaging pattern.

Problem

Common implementations of distributed solutions rely on remote invocation frameworks, such as those based on RPC technology. These communication systems establish persistent connections based on binary protocols to enable the exchange of data between units of logic.

Although efficient and reliable, they are primarily utilized within the boundaries of application environments and for select integration purposes. Positioning an RPC-based component as an enterprise resource with multiple potential consumers can lower its concurrency threshold because of the overhead associated with creating, sustaining, and terminating the required persistent RPC binary connections.

Solution

Messaging provides an alternative communications framework that does not rely on persistent connections. Instead, messages are transmitted as independent units of communication routed via the underlying infrastructure (Figure 18.1).

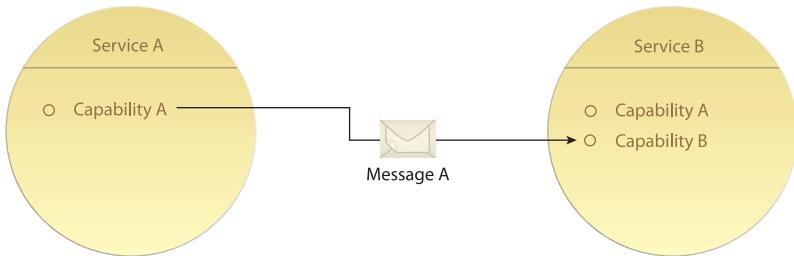


Figure 18.1

Services interact via the transmission of messages—self-contained units of communication.

Application

A messaging framework supported by the enterprise's technical environment needs to be implemented to an extent that it is capable of supporting service interaction requirements. Many established design patterns for messaging frameworks exist, most of which emerged from experience with enterprise integration platforms.

Impacts

Some messaging frameworks cannot provide an adequate level of QoS to support the high demands that can be placed on services positioned as reusable enterprise resources.

To fully enable the application of Capability Recomposition (526) and many of the supporting patterns, the message framework must provide a means of:

- guaranteeing the delivery of each message or guaranteeing a notification of failed deliveries
- securing message contents beyond the transport
- managing state and context data across a service activity
- transmitting messages efficiently as part of real-time interactions
- coordinating cross-service transactions

Without these types of extensions in place, the availability, reliability, and reusability of services will impose limitations that can undermine the strategic goals associated with SOA in general. As explained in the upcoming *Relationships* section, several, more specialized patterns address these individual issues.

Relationships

As one of the most fundamental design patterns in this catalog, Service Messaging ties directly into interoperability design considerations. The success of this pattern is therefore often dependent on the extent to which Canonical Protocol (150) and Canonical Schema (158) are applied within a given inventory.

Service Agent (543) forms a functional relationship with Service Messaging in that event-driven agent programs transparently intercept and process message contents. Messaging Metadata (538) is also closely related because it essentially extends the typical message to incorporate meta details.

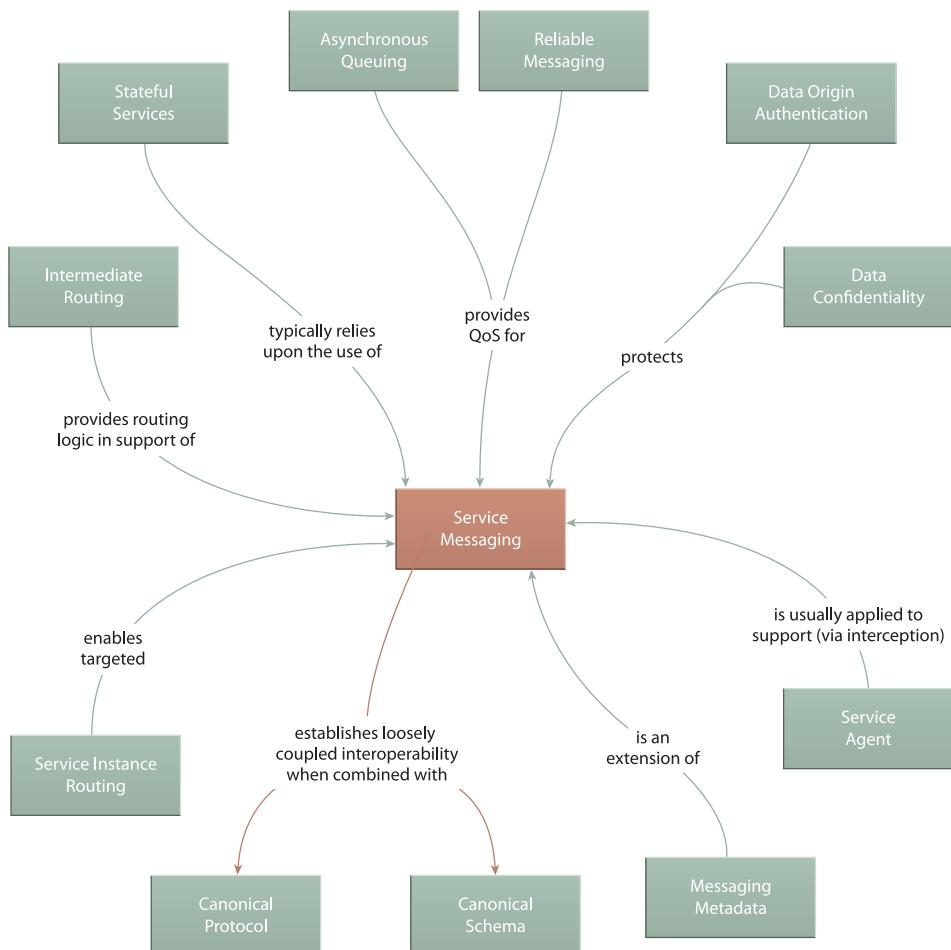


Figure 18.2

Service Messaging establishes the basis for many, more specialized communication and messaging-related patterns.

NOTE

As mentioned in Chapter 5, this pattern is related to several patterns documented in Hohpe and Woolf's book *Enterprise Integration Patterns*, including Message, Messaging, and Document Message and can further be linked to Message Channel and Message Endpoint. Numerous additional specialized messaging patterns documented in this book were established during the EAI era and can still help solve design problems in support of service-oriented solutions, especially in relation to enabling asynchronous message exchanges.

CASE STUDY EXAMPLE

Prior to the SOA initiative, most of the FRC's distributed solutions were based solely on RPC technology. This communications framework was very efficient and reliable but also posed several recurring challenges:

- Components established persistent connections that consumed excessive memory. At peak volume periods, server resource thresholds were regularly surpassed, introducing noticeable latency to all users.
- Most component interaction was based on the exchange of granular parameter data or entire records sets grouped into a proprietary binary format. The resulting communication requirements led to numerous roundtrips between clients and servers, further taxing resources and tightly binding components together.
- It was very difficult to change existing distributed designs due to the numerous cross-component dependencies that were formed. Communication patterns were rigid and primarily synchronous, leaving little opportunity to streamline interaction scenarios.

As part of this service delivery project and the overarching SOA initiative, the majority of the new services are being developed and implemented using Web service technology. Therefore, the use of messaging for inter-service communication is a natural requirement.

In order to leverage the many WS-* extensions that have been identified as key parts of the enterprise service-oriented architecture the FRC is planning, SOAP is chosen as the standard message format.

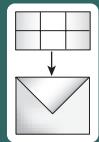
Although this introduces several new challenges associated with QoS concerns and different types of performance constraints, several improvements are also acknowledged:

- Due to the use of the connection-less HTTP protocol, the overhead associated with persistent binary connections is avoided. This noticeably frees server resources and allows for greater amounts of concurrent communications activity.
- Service interaction is document-centric, in that the data granularity of Web service operations is increased, allowing them to exchange larger bodies of data with each roundtrip. This minimizes traffic and further supports future scalability.
- Service governance benefits from less stringent cross-service dependencies, enabling compositions to be more easily reconfigured. Additionally, more creative and streamlined communication is made possible by combining synchronous and asynchronous message exchange patterns supported by the Web services framework.

The transition to a messaging-based communications framework is deemed worthwhile by the FRC because its fundamental characteristics are more in alignment with the positioning of services as recomposable IT assets.

Messaging Metadata

How can services be designed to process activity-specific data at runtime?



Problem	Because messaging does not rely on a persistent connection between service and consumer, it is challenging for a service to gain access to the state data associated with an overall runtime activity.
Solution	Message contents can be supplemented with activity-specific metadata that can be interpreted and processed separately at runtime.
Application	This pattern requires a messaging framework that supports message headers or properties.
Impacts	The interpretation and processing of messaging metadata adds to runtime performance overhead and increases service activity design complexity.
Principles	Service Loose Coupling, Service Statelessness
Architecture	Composition

Table 18.2

Profile summary for the Messaging Metadata pattern.

Problem

Persistent binary connections between a service and its consumer place various types of state and context data about the current service activity into memory, allowing routines within service capabilities to access this information as required.

Moving from RPC-based communication toward a messaging-based framework removes this option, as persistent connections are no longer available. This can place the burden of runtime activity management onto the services themselves.

Solution

State data, rules, and even processing instructions can be located within the messages. This reduces the need for services to share activity-specific logic (Figure 18.3).

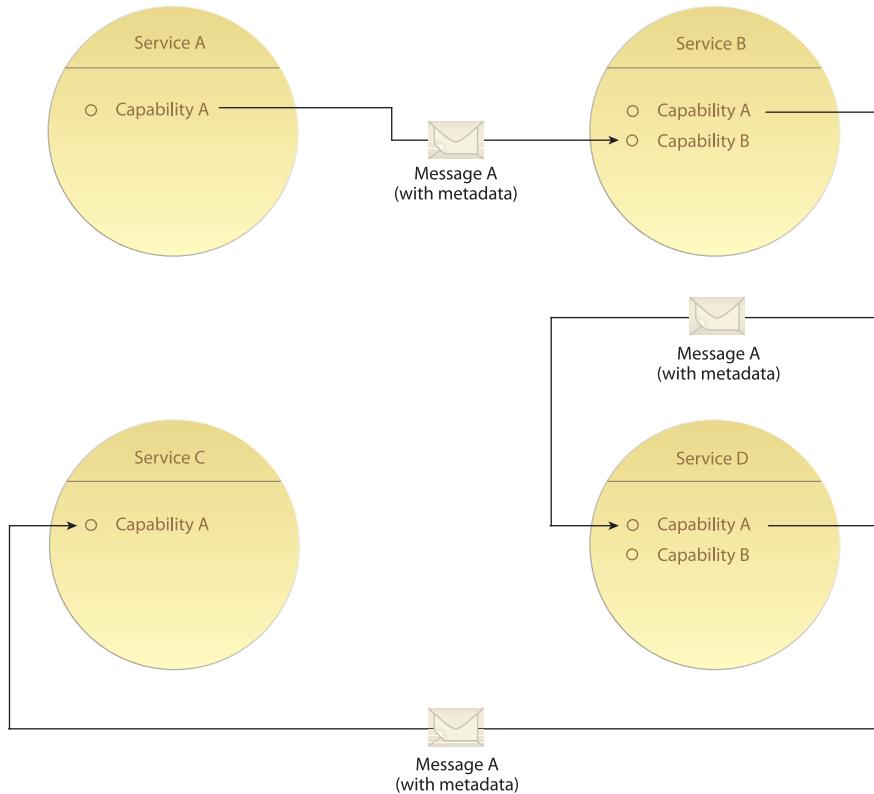


Figure 18.3

Messages equipped with metadata reduce the requirements for services to contain embedded, activity-specific solution logic.

Application

The messaging technology used for service communication needs to support message headers or properties so that the messaging metadata can be consistently located within a reserved part of the message document, as pointed out in Figure 18.4.

Platform-specific technologies, such as JMS, provide support for message headers and properties, as do Web service-related standards, such as SOAP. In fact, many types of messaging metadata have been standardized through the emergence of WS-* extensions that define industry standard SOAP header blocks, as demonstrated in a number of the case studies in this chapter.

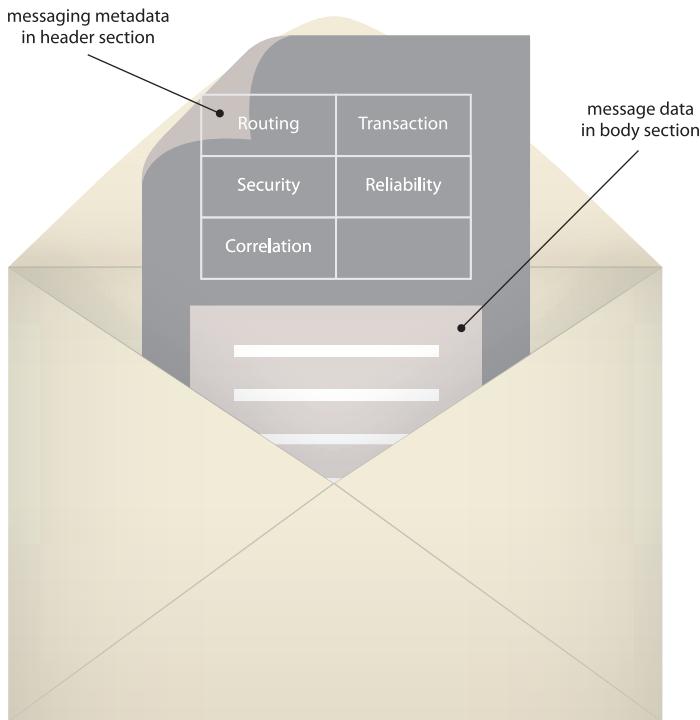


Figure 18.4

Header blocks allow for various types of metadata to accompany the message contents.

Impacts

Although overall memory consumption is lowered by avoiding a persistent binary connection, performance demands are increased by the requirement for services to interpret and process metadata at runtime. Agnostic services especially can impose more runtime cycles, as they may need to be outfitted with highly generic routines capable of interpreting and processing different types of messaging headers so as to participate effectively in multiple composition activities.

Due to the prevalence and range of technology standards that intrinsically support and are based on Messaging Metadata, a wide variety of sophisticated message exchanges can be designed. This can lead to overly creative and complex message paths that may be difficult to govern and evolve.

Relationships

This fundamental pattern can be seen as an extension of Service Messaging (533). Service compositions that rely on industry standard transaction management, security, routing and reliable messaging will utilize specialized implementations of this pattern, as represented by the variety of message-related patterns shown in Figure 18.5.

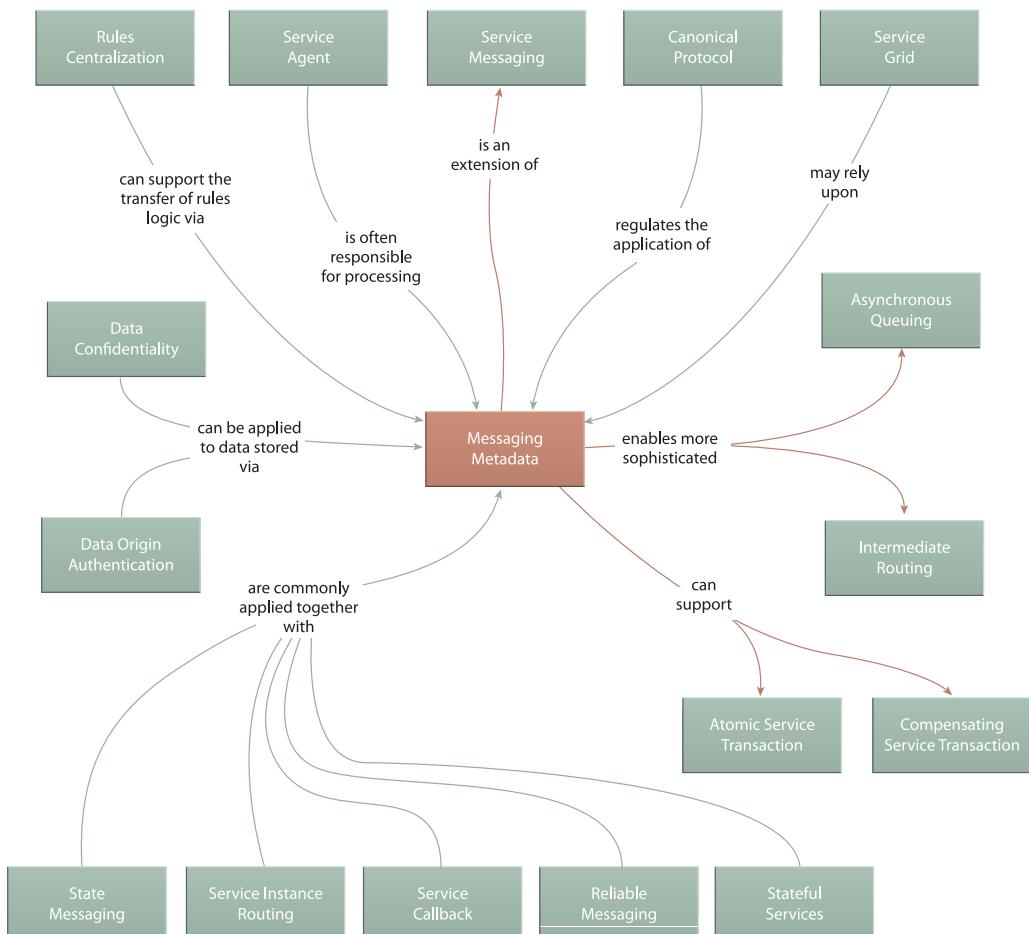


Figure 18.5

Messaging Metadata is commonly associated with patterns that provide extensions to composition architectures.

CASE STUDY EXAMPLE

When messages are routed through various intermediate services within the Cutit Saws environment, they are equipped with a metadata construct that establishes a correlation identifier. Cutit architects use the WS-Addressing `MessageID` element to express correlation values via industry standard header blocks within their SOAP messages, as follows:

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"  
    xmlns:wsa="http://schemas.xmlsoap.org/ws/  
    2004/08/addressing">  
    <Header>  
        <wsa:MessageID>  
            uuid:938993-226  
        </wsa:MessageID>  
    </Header>  
    <Body>  
        ...  
    </Body>  
</Envelope>
```

Example 18.1

The `MessageID` construct defines a SOAP header block as per the conventions established in the WS-Addressing specification.

NOTE

For more examples of how this pattern is applied with Web services and WS-* standards, see Chapters 6, 7, and 17 in *Service-Oriented Architecture: Concepts, Technology, and Design* and Chapters 4, 11, 15, 18, and 19 in *Web Service Contract Design and Versioning for SOA*.

Service Agent

How can event-driven logic be separated and governed independently?



Problem	Service compositions can become large and inefficient, especially when required to invoke granular capabilities across multiple services.
Solution	Event-driven logic can be deferred to event-driven programs that don't require explicit invocation, thereby reducing the size and performance strain of service compositions.
Application	Service agents can be designed to automatically respond to predefined conditions without invocation via a published contract.
Impacts	The complexity of composition logic increases when it is distributed across services, and event-driven agents and reliance on service agents can further tie an inventory architecture to proprietary vendor technology.
Principles	Service Loose Coupling, Service Reusability
Architecture	Inventory, Composition

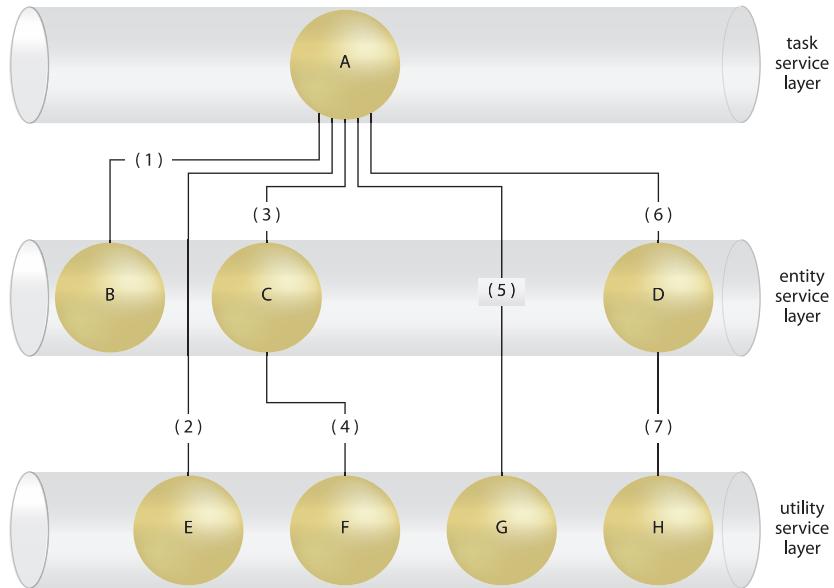
Table 18.3

Profile summary for the Service Agent pattern.

Problem

Service composition logic consists of a series of service invocations; each invocation enlisting a service to carry out a segment of the overall parent business process logic. Larger business processes can be enormously complex, especially when having to incorporate numerous “what if” conditions via compensation and exception handling sub-processes. As a result, service compositions can grow correspondingly large (Figure 18.6).

Furthermore, each service invocation comes with a performance hit resulting from having to explicitly invoke and communicate with the service itself. The performance of larger compositions can suffer from the collective overhead of having to invoke multiple services to automate a single task.

**Figure 18.6**

A service sequentially composing several others to carry out a particular task.

Solution

Service logic that is triggered by a predictable event can be isolated into a separate program especially designed for automatic invocation upon the occurrence of the event (Figure 18.7). This reduces the amount of composition logic that needs to reside within services and further decreases the quantity of services (or service invocations) required for a given composition.

Application

The event-driven logic is implemented as a service agent—a program with no published contract that is capable of intercepting and processing messages at runtime. Service agents are typically lightweight programs with modest footprints and generally contain common utility-centric processing logic.

For example, vendor runtime platforms commonly provide “system-level” agents that carry out utility functions such as authentication, logging, and load balancing. Service agents can also be custom-developed to provide business-centric and/or single-purpose logic as well.

As first introduced in Figure 4.14 in the *Service Architecture* section of Chapter 4, the message processing logic that is a natural part of any Web service implementation actually consists of a series of system (and perhaps custom) service agents that collectively carry out necessary runtime actions.

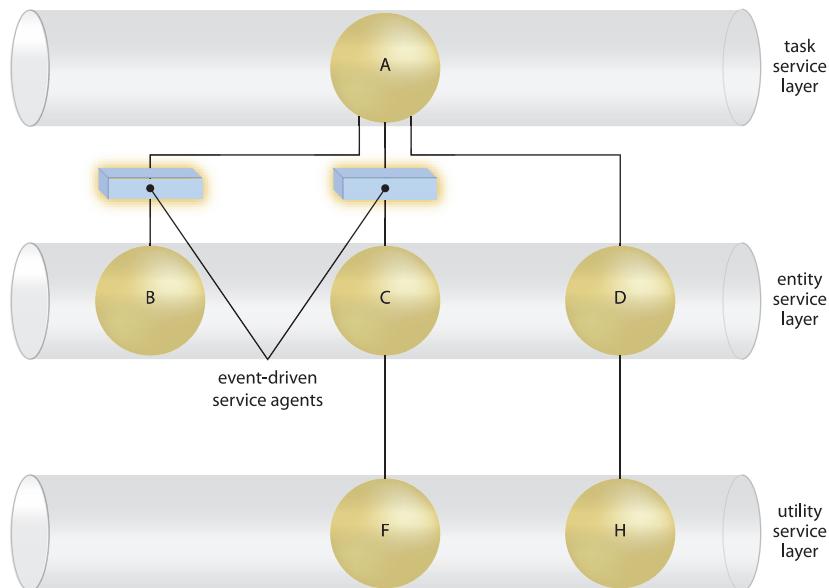


Figure 18.7

Two service agents replace the need for the explicit invocation of utility services E and G. By deferring common logic to service agents, the overall quantity of explicitly invoked services decreases.

NOTE

Service agents are most commonly deployed to facilitate inter-service communication, but they can also be utilized within service architectures. In fact, intra-service use of agents can avoid some of the vendor dependency issues that arise with inter-service agent usage, as explained next in the *Impacts* section.

Impacts

Event-driven agents provide yet another layer of abstraction to which multiple service compositions can form dependencies. Although the perceived size of the composition may be reduced, the actual complexity of the composition itself does not decrease. Composition logic is simply more decentralized as it now also encompasses service agents that automatically perform portions of the overall task.

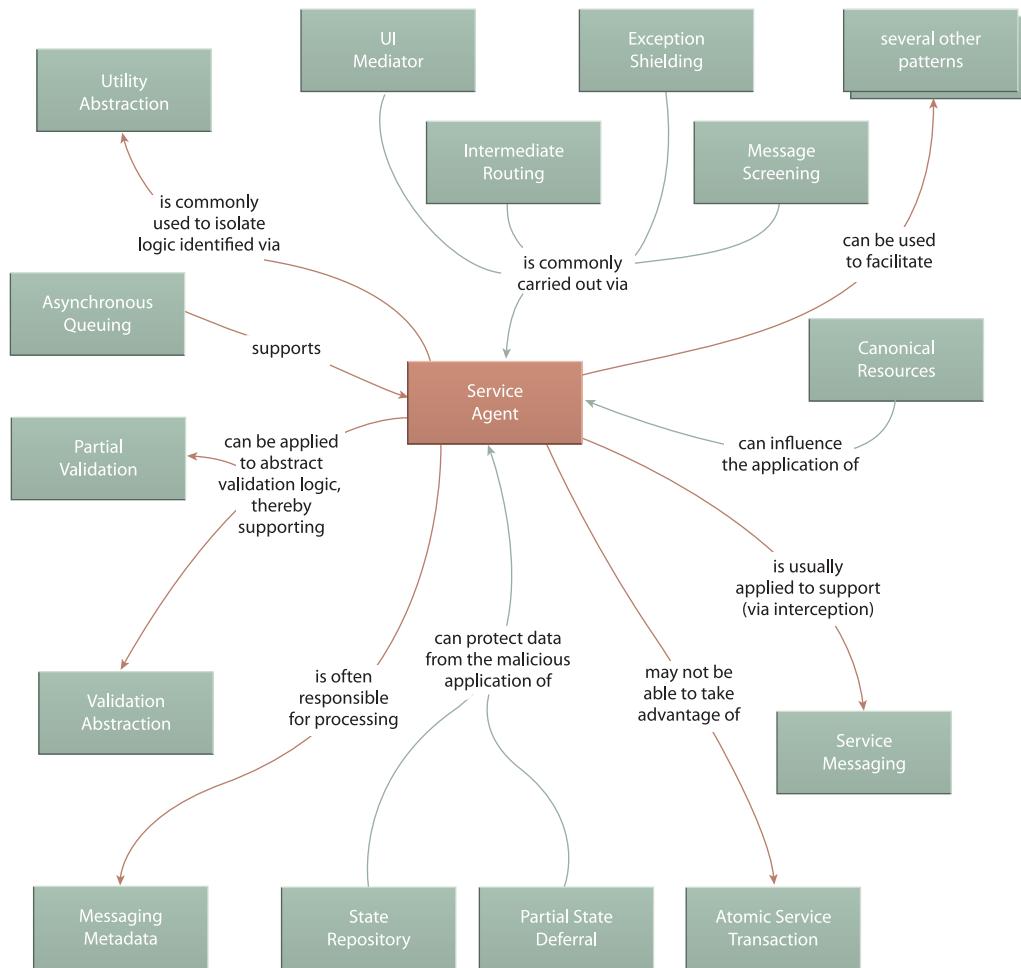
Overuse of this design pattern can result in an inventory architecture that is difficult to build services for. With too many service agents transparently performing a range of functions, it can become too challenging to design composition architectures that take all possible agent-related processing scenarios into account. Furthermore, some service agent programs may end up conflicting with other service agents or other service logic.

Governance can also become an issue in that service agents will need to be owned and maintained by a separate group that needs to understand the inventory-wide impacts of any changes made to agent logic. For example, system service agents can be subject to behavioral changes as a result of runtime platform upgrades. An agent versioning system will be further required to address these challenges.

Relationships

The event-driven programs created as a result of applying this pattern become a common and intrinsic part of service-oriented inventory architectures. The type of logic they encapsulate is comparable to utility logic, and therefore similar design considerations are most commonly applied. Either way, Service Agent's most fundamental relationships are with Service Messaging (533) and Messaging Metadata (538).

As previously mentioned, the overuse of this pattern can lead to an undesirably high level of dependency on a vendor platform. This can be due to the need to build custom service agents with proprietary programming languages or because services rely too heavily on the proprietary agents provided by vendor runtime environments. Canonical Resources (237) can alleviate this, but it does not directly regulate the *quantity* of produced agents.

**Figure 18.8**

This diagram reveals the extent to which Service Agent can be utilized when applying SOA design patterns. Note the "several other patterns" block at the top right indicating that further relationships exist.

NOTE

Service Agent is closely related to Event-Driven Consumer (Hohpe, Woolf).

CASE STUDY EXAMPLE

Within an average day at one of the Alleywood mills, trucks haul loads of raw trees to a primary bay where the amount of usable wood is assessed and then unloaded for further processing. This process is tracked in that the arrival and departure of every truck is logged, along with each received load.

As workers record this information via mobile hand-held devices with keypads, the data is processed by the Truck and Load services, which are commonly composed by a separate task service responsible for specific types of load deliveries.

As part of a typical service composition, the Truck service is required to invoke and send data to the Load service depending on the size and nature of a given load. Alleywood architects have positioned two custom service agents that facilitate this exchange by providing supplementary logging and validation functions (Figure 18.9).

The Request Logger agent captures any data related to loads that are below a minimum quantity or contain a high percentage of “non-processable” raw materials. These are written to a separate database and form the basis of analytical statistics used by Alleywood process engineers. The Header Validation agent simply validates a set of custom headers used for load and truck tracking purposes.

Both service agents perform functions that are common to other services. The Request Logger agent, for example, has several rules built into it, enabling it to silently log a series of what are considered “abnormal conditions.” Similarly, the Header Validation agent is designed to validate all custom Alleywood headers.

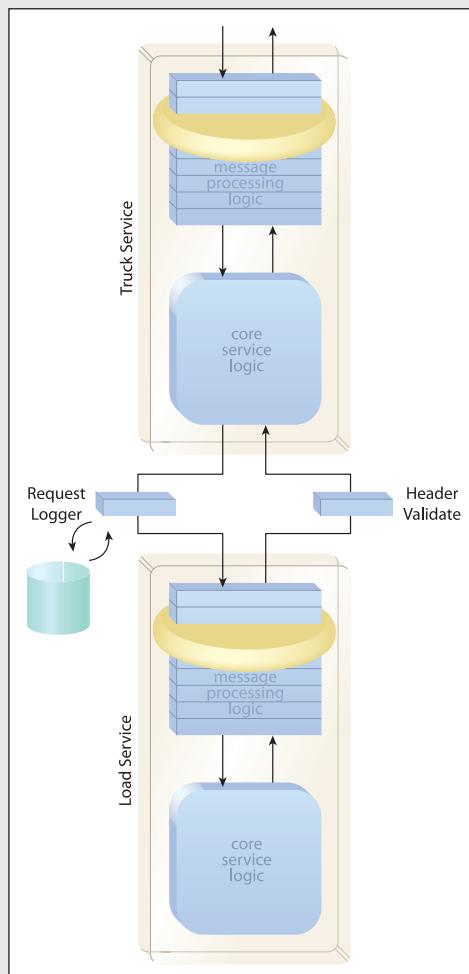


Figure 18.9

Two service agents are employed in a request-response data exchange.

Intermediate Routing

By Mark Little, Thomas Rischbeck, Arnaud Simon



How can dynamic runtime factors affect the path of a message?

Problem	The larger and more complex a service composition is, the more difficult it is to anticipate and design for all possible runtime scenarios in advance, especially with asynchronous, messaging-based communication.
Solution	Message paths can be dynamically determined through the use of intermediary routing logic.
Application	Various types of intermediary routing logic can be incorporated to create message paths based on message content or runtime factors.
Impacts	Dynamically determining a message path adds layers of processing logic and correspondingly can increase performance overhead. Also the use of multiple routing logic can result in overly complex service activities.
Principles	Service Loose Coupling, Service Reusability, Service Composability
Architecture	Composition

Table 18.4

Profile summary for the Intermediate Routing pattern.

Problem

A service composition can be viewed as a chain of point-to-point data exchanges between composition participants. Collectively, these exchanges end up automating a parent business process.

The message routing logic (the decision logic that determines how messages are passed from one service to another) can be embedded within the logic of each service in a composition. This allows for the successful execution of *predetermined* message paths. However, there may be unforeseen factors that are not accounted for in the embedded routing logic, which can lead to unanticipated system failure.

For example:

- The destination service a message is being transmitted to is temporarily (or even permanently) unavailable.

- The embedded routing logic contains a “catch all” condition to handle exceptions, but the resulting message destination is still incorrect.
- The originally planned message path cannot be carried out, resulting in a rejection of the message from the service’s previous consumer.

Figure 18.10 illustrates these scenarios.

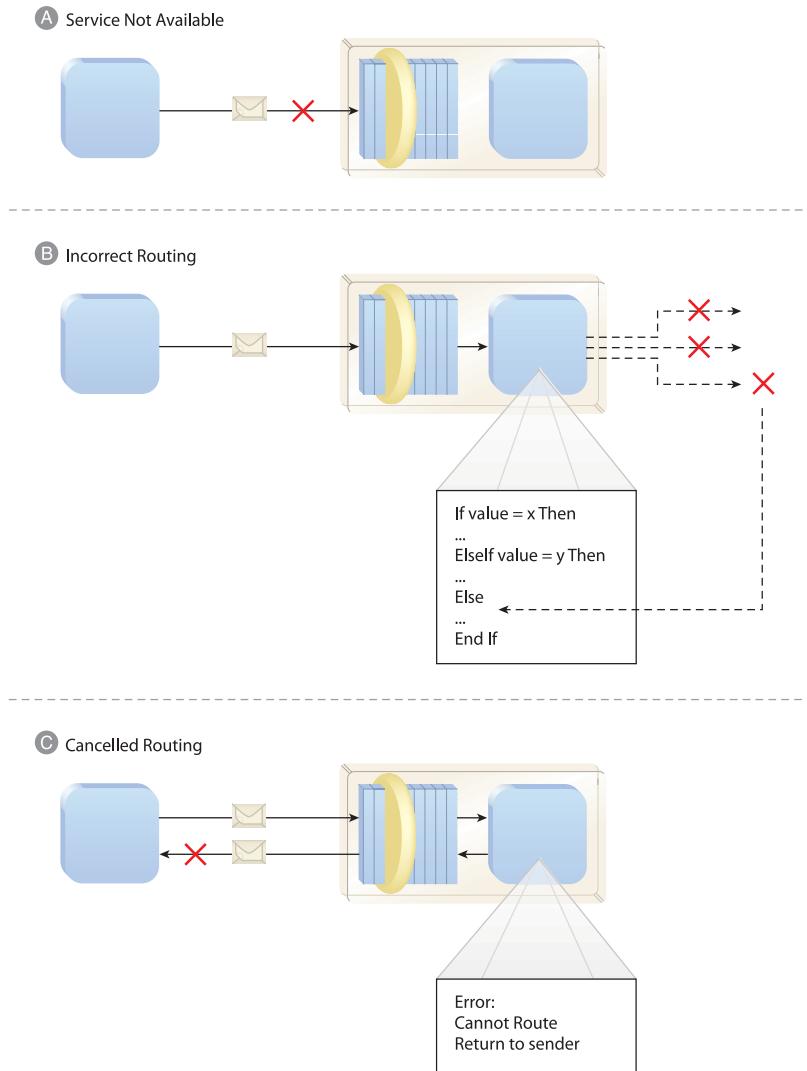


Figure 18.10

A message transmission fails because the service is not available (A). Internal service routing logic is insufficient and ends up sending the message to the wrong destination (B). Internal service logic is incapable of routing the message and simply rejects it (C), effectively terminating the service activity.

Alternatively, there may simply be functional requirements that are dynamic in nature and for which services cannot be designed in advance.

Solution

Generic, multi-purpose routing logic can be abstracted so that it exists as a separate part of the architecture in support of multiple services and service compositions. Most commonly this is achieved through the use of event-driven service agents that transparently intercept messages and dynamically determine their paths (Figure 18.11).

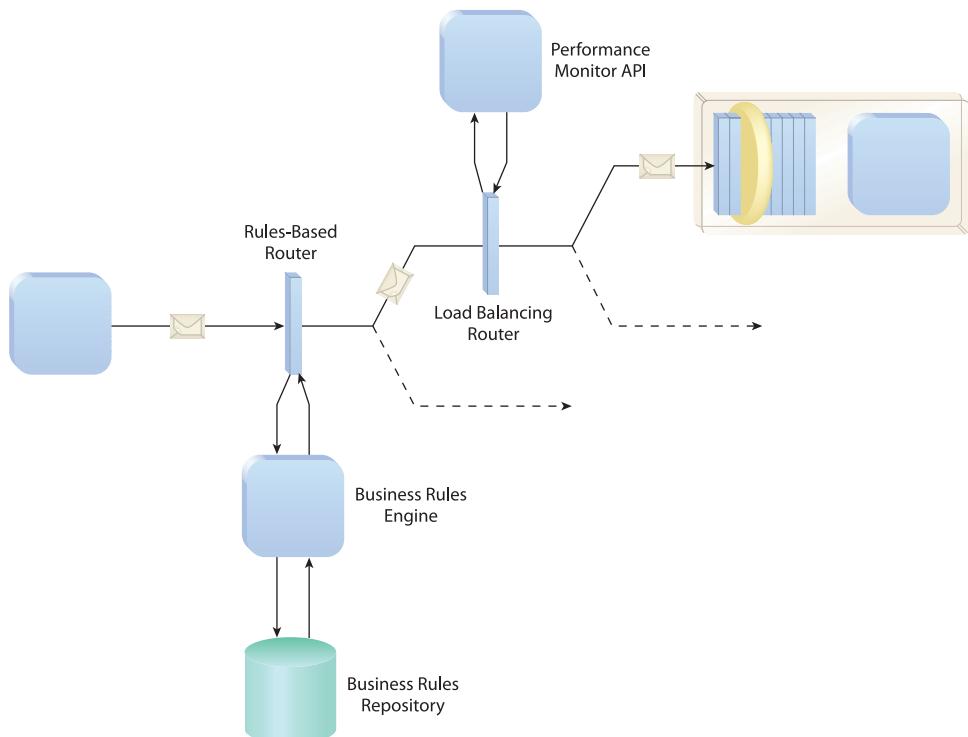


Figure 18.11

A message passes through two router agents before it arrives at its destination. The Rules-Based Router identifies the target service based on a business rule that the agent dynamically retrieves and interprets, as a consequence of Rules Centralization (216). The Load Balancing Router then checks current usage statistics for that service before it decides which instance or redundant implementation of the service to send the message to.

Application

This pattern is usually applied as a specialized implementation of Service Agent (543). Routing-centric agents required to perform dynamic routing are often provided by messaging middleware and are a fundamental component of ESB products. These types of out-of-the-box agents can be configured to carry out a range of routing functions. However, the creation of custom routing agents is also possible and not uncommon, especially in environments that need to support complex service compositions with special requirements.

Common forms of routing functionality include:

- *Content-Based Routing* – Essentially, this type of routing determines a message's path based on its contents. Content-based routing can be used to model complex business processes and provide an efficient way to recompose services on the fly. Such routing decisions may need to involve access to a business rules engine to accurately assess message destinations.
- *Load Balancing* – This form of routing agent has become an important part of environments where concurrent usage demands are commonplace. A load balancing router is capable of directing a message to one or more identical service instances in order to help the service activity be carried out as efficiently as possible.
- *1:1 Routing* – In this case, the routing agent is directly wired to a single physical service at any point in time. When messages arrive, the agent is capable of routing them to different service instances or redundant service implementations. This accommodates standard fail-over requirements and allows services to be maintained or upgraded without risking “disruption of service” to consumers.

Regardless of the nature of the routing logic, it is desirable to be able to update and modify routing parameters dynamically—ideally even by business analysts so that they can adapt and control the business logic in real-time. This is particularly important when business logic is subject to frequent change. If changes are *extremely* frequent, it can be further beneficial to model the routing logic through the extraction of complex *business rules* that describe declarative logic on top of the message content and use the outcome to make the routing decision.

A more frugal alternative is to employ content-based routing using XPath or XQuery expressions. However, these languages require technically more involved personnel for their control and maintenance.

NOTE

While event-driven agents represent the most common implementation of this pattern, routing logic can also be incorporated into actual intermediary services that process and forward messages based on the same factors as those previously listed.

Impacts

The usage of routing agents allows the automation of complex decisions and the quick adaptation to changing business requirements. However, the complexity and flexibility of incorporating intermediate routing logic into composition architectures is not without disadvantages:

- Dynamic modification of routing rules at runtime can introduce the risk of having previously untested logic set into production. If possible, routing rule-set changes should first be put through a conventional staging process.
- Dynamic routing paths can be elaborate and therefore difficult to manage and update, leading to a risk of unexpected failure conditions. A centralized routing rule management system can help alleviate the risk of introducing potential points of failure.
- Physically separated routing logic will naturally add performance overhead when compared to direct service-to-service communication, where the routing logic is embedded within the consumer program.

Additionally, security can be a concern when applying this pattern. You may want to control who will and will not process a message containing sensitive data. An inventory architecture with many built-in intermediate routing agents can provide native functionality that conflicts with some security requirements.

Relationships

Routing functionality is a fundamental part of messaging frameworks and can therefore be associated with most messaging-related design patterns. The separation of metadata provided by Messaging Metadata (538) allows for the more advanced forms of routing described earlier in the *Application* section. Also when implementing routing logic with Service Agent (543), Canonical Resources (237) can influence the platform and technologies used to build and host the agent programs.

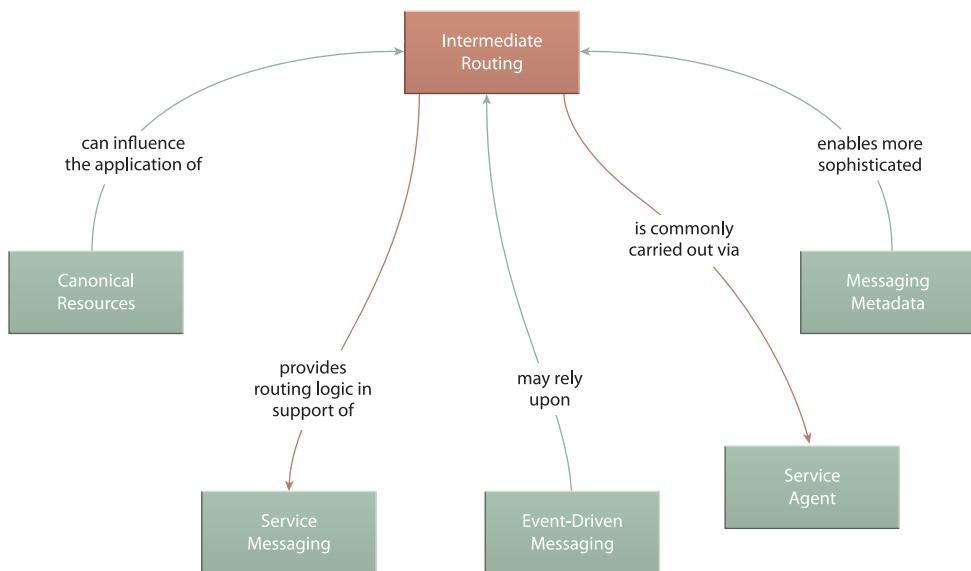


Figure 18.12

Intermediate Routing provides specialized, agent-related processing in support of message transmissions and therefore has relationships with several messaging patterns.

Because of their messaging-centric feature-sets, ESB platforms are fully expected to carry out routing functionality in support of sophisticated service activity process. Intermediate Routing is therefore one of the three core patterns that represent Enterprise Service Bus (704).

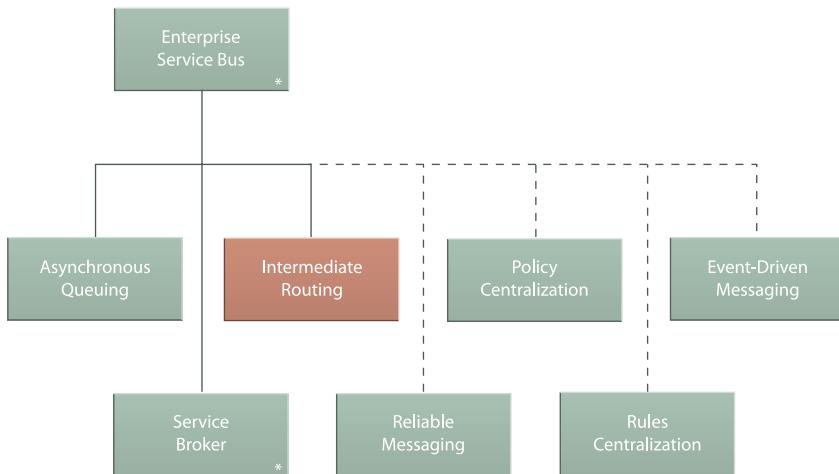


Figure 18.13

Intermediate Routing is one of the patterns that comprise Enterprise Service Bus (704).

NOTE

Depending on how it is applied, the runtime routing logic established by Intermediate Routing is comparable to Content-Based Router (Hohpe, Woolf), Dynamic Router (Hohpe, Woolf), and Message Router (Hohpe, Woolf). Several more message routing patterns are described in the *Enterprise Integration Patterns* catalog. Intermediate Routing highlights the most common routing options used for service composition architectures.

CASE STUDY EXAMPLE

The Flight Plan Validation service described later in the example for Data Format Transformation (681) forwards flight plan documents after they have successfully been transformed and validated.

As part of the FRC Aerial Firefighting Coordination business process, content-based routers are employed to route these documents through the appropriate service, based on the following factors:

- the SOAP body of the received flight plan messages
- SOAP headers that may indicate limited availability or delayed arrival of aircrafts
- the severity level of the current wildfire
- the location to which the aircrafts will need to travel (usually the location of the fire)

The routing intelligence is complex and even subject to further changes due to international regulatory compliance requirements. As shown in Figure 18.14, all of these factors determine the ultimate decision as to which service to forward the message to.

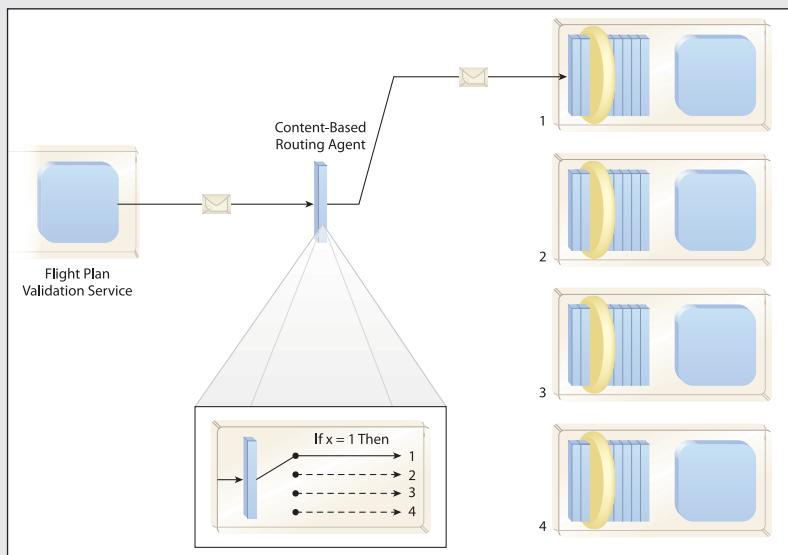
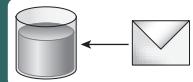


Figure 18.14

The content-based routing agent receives the message from the Flight Plan Validation service and then dynamically determines where to route it to. Note that in this case, the service may be sending the message to a logical address that is processed by the agent, which then forwards it to the appropriate physical address.

State Messaging

By Anish Karmarkar



How can a service remain stateless while participating in stateful interactions?

Problem	When services are required to maintain state information in memory between message exchanges with consumers, their scalability can be comprised, and they can become a performance burden on the surrounding infrastructure.
Solution	Instead of retaining the state data in memory, its storage is temporarily delegated to messages.
Application	Depending on how this pattern is applied, both services and consumers may need to be designed to process message-based state data.
Impacts	This pattern may not be suitable for all forms of state data, and should messages be lost, any state information they carried may be lost as well.
Principles	Standardized Service Contract, Service Statelessness, Service Composability
Architecture	Composition, Service

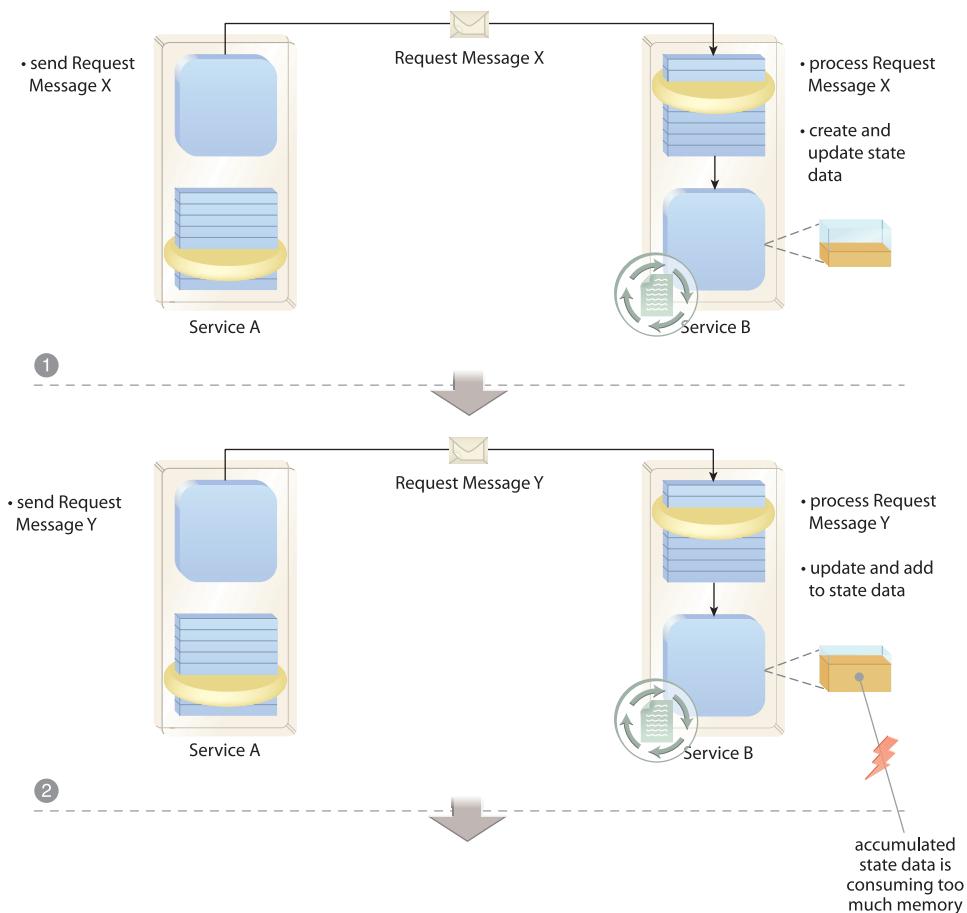
Table 18.5

Profile summary for the State Messaging pattern.

Problem

Services are sometimes required to be involved in runtime activities that span multiple message exchanges. In these cases, a service may need to retain state information until the overarching task is completed. This is especially common with services that act as composition controllers.

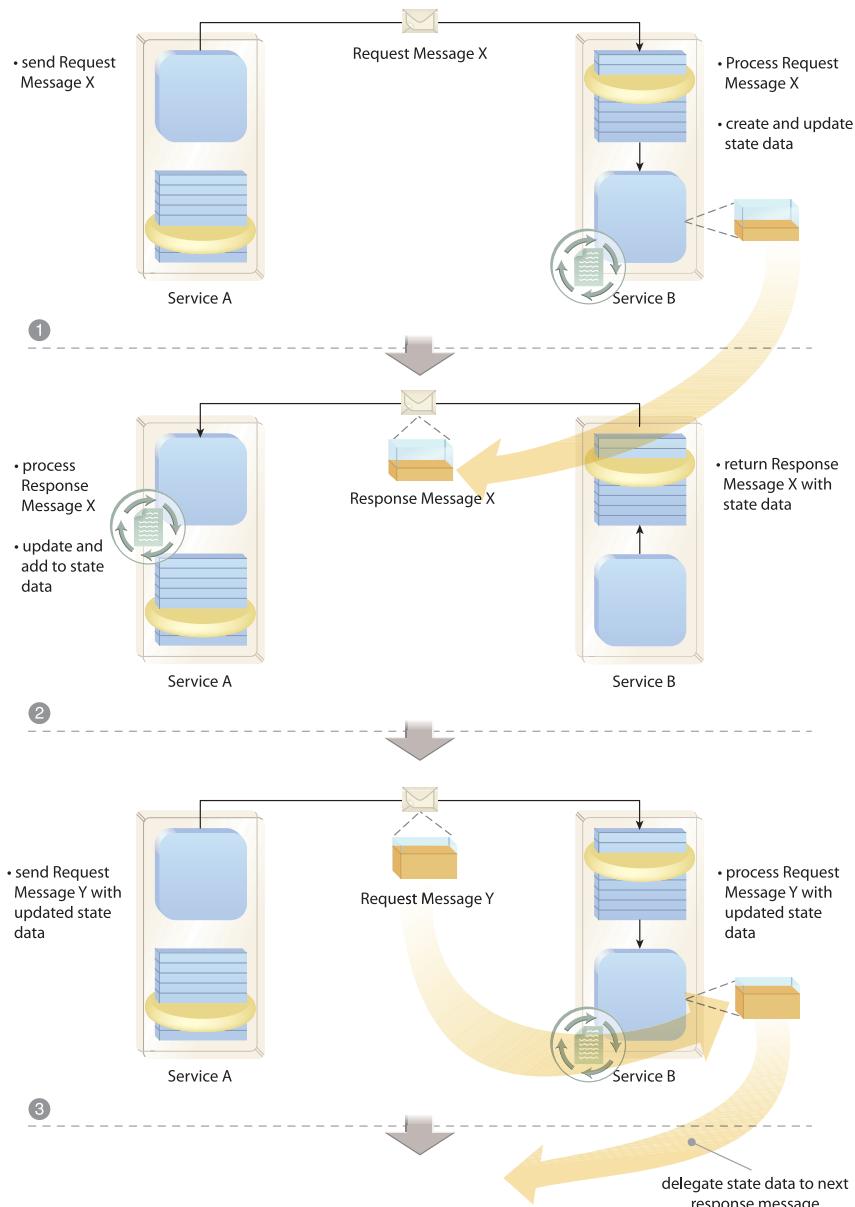
By default, services are often designed to keep this state data in memory so that it is easily accessible and essentially remains alive for as long as the service instance is active (Figure 18.15). However, this design approach can lead to serious scalability problems and further runs contrary to the Service Statelessness design principle.

**Figure 18.15**

This figure shows just a part of a larger conversational exchange between two services. Service A, acting as a service consumer, issues a Request Message X to Service B (1). Service B creates the necessary data structures to maintain the state associated with processing Request Message X and updates the data structures after processing is completed. Service A then issues another request to Service B (2), which Service B then processes, resulting in an update of the state data that also increases the quantity of state data Service B must retain (which, ultimately, results in scalability problems).

Solution

Instead of the service maintaining state data in memory, it moves the data to the message. During a conversational interaction, the service retrieves the latest state data from the next input message (Figure 18.16).

**Figure 18.16**

Service A, acting as a service consumer, issues Request Message X to Service B (1). Service B creates the necessary data structures to maintain the necessary state and updates the data structures after processing this message. Service B then adds the state data to Response Message X, which it then returns back to Service A (2). Service A processes the response and then generates Request Message Y containing updated state data, which is then received and processed by Service B (3).

Application

There are two common approaches to applying this pattern, both of which affect how the service consumer relates to the state data:

- The consumer retains a copy of the latest state data in memory and only the service benefits from delegating the state data to the message. This approach is suitable for when this pattern is implemented using WS-Addressing, due to the one-way conversational nature of Endpoint References (EPRs).
- Both the consumer and the service use messages to temporarily off-load state data. This two-way interaction with state data may be appropriate when both consumer and service are actual services within a larger composition. This technique can be achieved using custom message headers.

With either approach, this pattern requires that the messaging infrastructure be capable of distinguishing between message body content (or payload data) and supplementary meta-data commonly stored in message headers. Whereas with WS-Addressing these message headers can be processed by many modern messaging products and platforms, the custom header approach requires extra custom development effort.

It is important to note that both techniques introduce the need for proprietary service logic. While WS-Addressing standardizes the EPR wrapper elements used to house state data, it does not standardize the expression of the state data itself. When using custom headers, the need for proprietary processing logic required to extract and process state data from messages will span to both consumer and service.

NOTE

For examples of pre-defined SOAP headers that are suitable for sophisticated, two-way conversational message exchanges, view the WS-Context specification accessible via SOASpecs.com.

Impacts

When following the two-way model with custom headers, messages that are lost due to runtime failure or exception conditions will further lose the state data, thereby placing the overarching task in jeopardy.

It is also important to consider the security implications of state data placed on the messaging layer. For services that handle sensitive or private data, the corresponding state information should either be suitably encrypted and/or digitally signed, and it is not uncommon for the consumer to not gain access to protected state data.

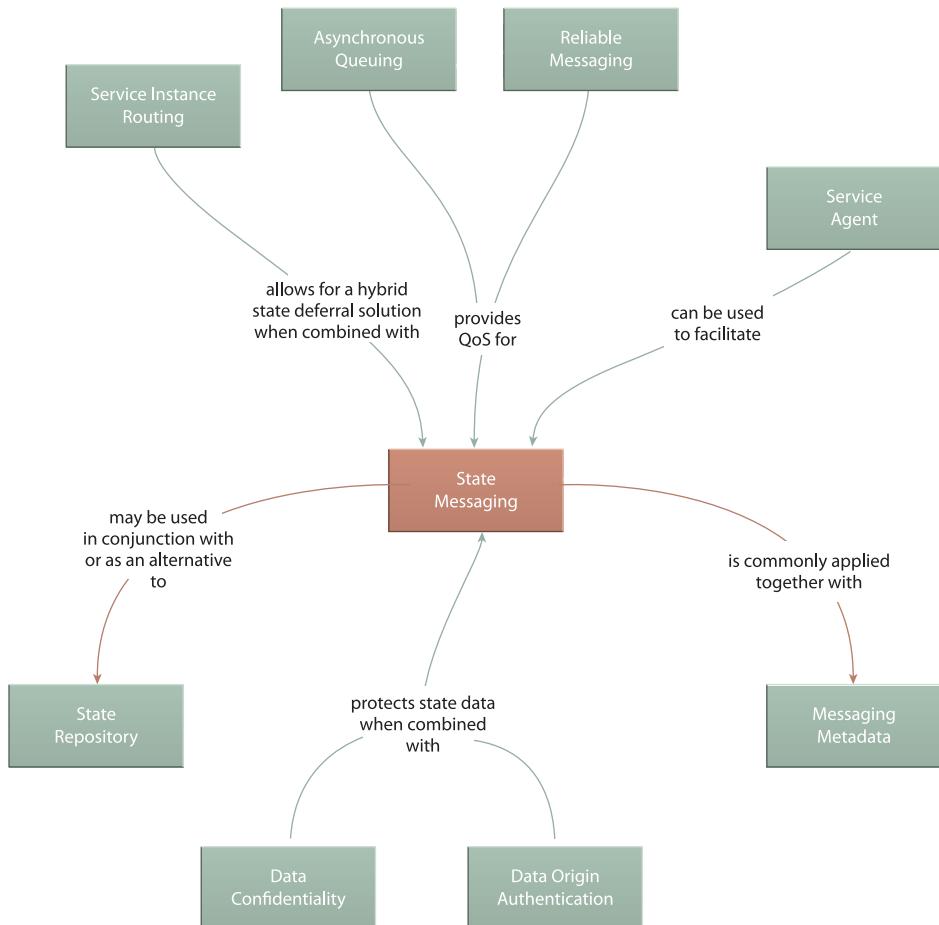
Furthermore, because this pattern requires that state data be stored within messages that are passed back and forth with every request and response, it is important to consider the size of this information and the implications on bandwidth and runtime latency. As with other patterns that require new infrastructure extensions, establishing inventory-wide support for State Messaging will introduce cost and effort associated with the necessary infrastructure upgrades.

Relationships

State Messaging is based on Service Messaging (533) and further utilizes Messaging Metadata (538) to represent state information within header blocks. This pattern can also be used in conjunction with Service Instance Routing (574) in such a way that only part of the state is maintained by the consumer, while the other part is managed by the service. This can lead to reduced message sizes and memory requirements.

Asynchronous Queuing (582) and Reliable Messaging (592) can be further utilized to provide increased robustness within the underlying infrastructure so that the state data remains protected against runtime failures and errors, even while in transit.

Also, as mentioned earlier, the use of state data with security requirements may demand that this pattern be combined with Data Confidentiality (641) and/or Data Origin Authentication (649).

**Figure 18.17**

State Messaging is associated with other message patterns but also those that provide often required quality-of-service and security extensions.

CASE STUDY EXAMPLE

In order to reduce operational costs at the FRC, management has decided to look for opportunities to downsize infrastructure. Several meetings with IT staff have generated some ideas, one of which is to simply shift some of the processing responsibilities out of the FRC enterprise and onto partner organizations. One application of this approach relates to the FRC's approval process for industrial machinery that forestry companies are required to undergo before they can purchase and put new lumber processing machines to use.

Up until now, the FRC has automated these approvals so that all parts of the approval process are carried out by the Approval service. This includes cases where all or some of the process steps are completed by partners.

Each occurrence of the process generates an application document that stores specifications and other required details about the machinery. These documents can become relatively large, and the FRC receives many of these applications daily. As a result, they are required to store large amounts of information, a good portion of which is state data that corresponds to documents that are part of incomplete approval processes.

In an effort to delegate processing and infrastructure requirements to partners, it is decided to re-engineer the Approval service so that it supports State Messaging. Specifically, by moving the state associated with a given approval process to the message layer, the responsibility for storing and maintaining approval documents can be delegated to the partner consumer applications until such a time that the process is completed (at which point the documents become part of the FRC archive).

At first, FRC architects build a prototype based on the use of custom headers with the intention of providing Alleywood and other partners the opportunity to delegate state data to messages as well.

After exploring scenarios involving this prototype further, the architects realize that they cannot proceed with this design due to a firm requirement that the FRC maintain control of state data during external interaction with partners. Allowing partners to access and even modify the status of an application, for example, would be strictly prohibited. As a result, the architects decide to standardize on the use of reference parameters provided by WS-Addressing Endpoint References (EPRs).

The following example shows a request message sent to the FRC that initiates the approval process:

```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://www.w3.org/2005/08/addressing"
  xmlns:frc="http://frc/ns/approval">
  <Header>
    <wsa:To>
      http://frc/submitApp
    </wsa:To>
    ...
  </Header>
  <Body>
    <frc:ApprovalStep1>
```

```
    ...
  </frc:ApprovalStep1>
</Body>
</Envelope>
```

Example 18.2

A SOAP message containing an application for approval. This message is the first in a series of messages that must be exchanged to complete the approval process.

The next example displays the corresponding response message generated by the FRC. It contains the `frc:ServiceEPR` construct (representing the EPR for the service), which includes a reference parameter element called `frc:AppApprovalState` that contains the status of the application (among other state details):

```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope"
  xmlns:frc="http://frc/ns/approval"
  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <Header>
    ...
  </Header>
  <Body>
    <!-- Response from FRC -->
    <frc:ApprovalStep1Response>
      <frc:ServiceEPR>
        <wsa:Address>
          http://frc/submitApp
        </wsa:Address>
        <wsa:ReferenceParameters>
          <frc:AppApprovalState>
            ...
          <frc:AppApprovalState>
            ...
          </frc:AppApprovalState>
        </wsa:ReferenceParameters>
      </frc:ServiceEPR>
      ...
    </frc:ApprovalStep1Response>
  </Body>
</Envelope>
```

Example 18.3

The FRC's response message that provides the state data located in the `wsa:ReferenceParameters` construct that is further wrapped within the `frc:ServiceEPR` construct, which represents a new EPR for the Approval service. Note that this message does not indicate the final approval but only approval from one of several FRC departments.

After receiving the message from Example 18.3, the partner organization issues the message shown in Example 18.4 to the destination within the FRC provided by the preceding message's EPR. In this case the state data is located in the `frc:AppApprovalState` construct in the SOAP header (as required by WS-Addressing):

```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://www.w3.org/2005/08/addressing"
  xmlns:frc="http://frc/ns/approval">
  <Header>
    <wsa:To>
      http://frc/submitApp
    </wsa:To>
    <frc:AppApprovalState wsa:IsReferenceParameter="true">
      ...
    </frc:AppApprovalState>
  </Header>
  <Body>
    <!-- Message sent for approval step 2 -->
    <frc:ApprovalStep2>
      ...
    </frc:ApprovalStep2>
  </Body>
</Envelope>
```

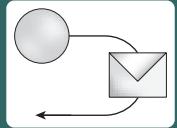
Example 18.4

The second message sent to the FRC for Step 2 of the approval process provides state data within the reference parameter `frc:AppApprovalState` as a SOAP header block. This header is used by the FRC service to recreate the current state of the application.

Service Callback

By Anish Karmarkar

How can a service communicate asynchronously with its consumers?



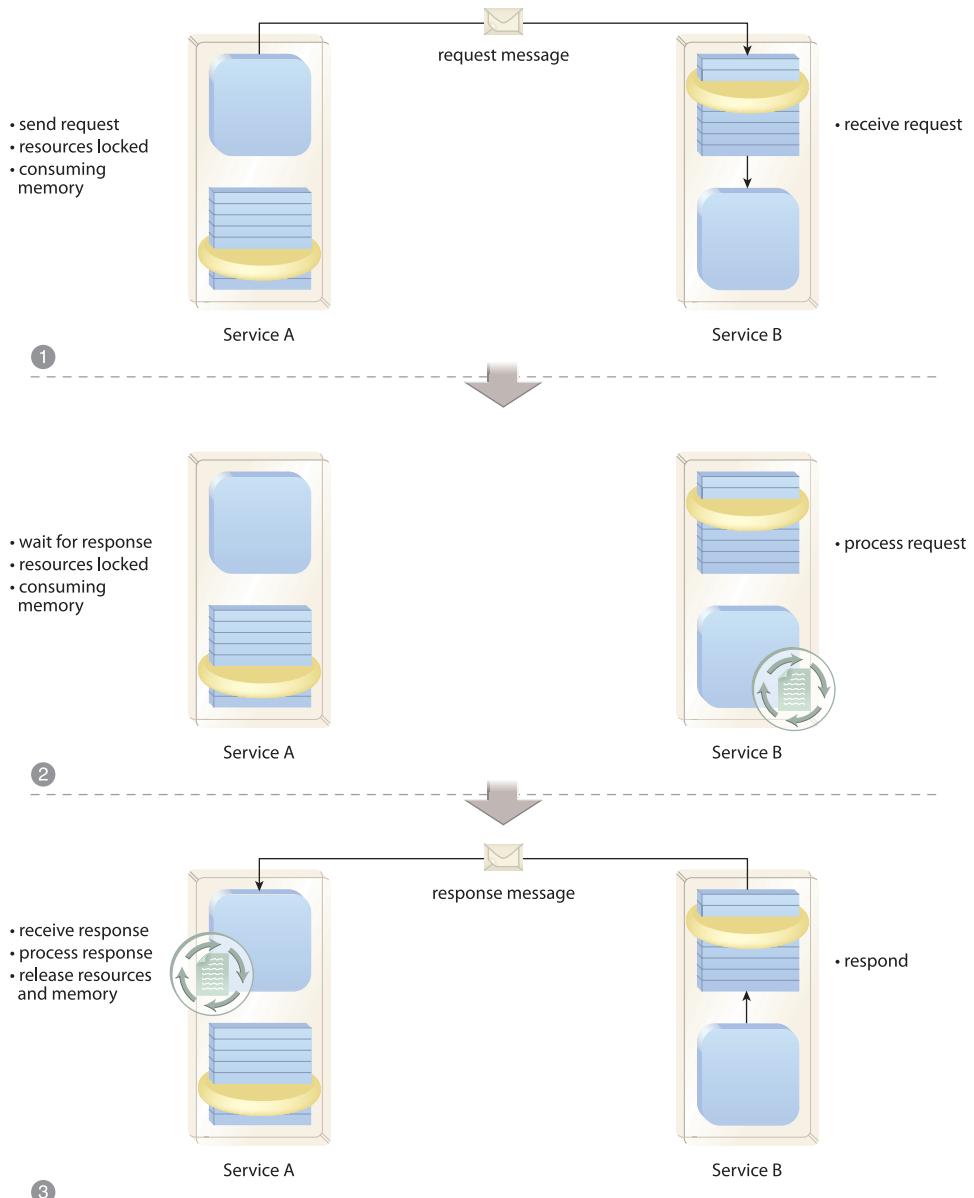
Problem	When a service needs to respond to a consumer request through the issuance of multiple messages or when service message processing requires a large amount of time, it is often not possible to communicate synchronously.
Solution	A service can require that consumers communicate with it asynchronously and provide a callback address to which the service can send response messages.
Application	A callback address generation and message correlation mechanism needs to be incorporated into the messaging framework and the overall inventory architecture.
Impacts	Asynchronous communication can introduce reliability concerns and can further require that surrounding infrastructure be upgraded to fully support the necessary callback correlation.
Principles	Standardized Service Contract, Service Loose Coupling, Service Composability
Architecture	Inventory, Service, Composition

Table 18.6

Profile summary for the Service Callback pattern.

Problem

When service logic requires that a consumer request be responded to with multiple messages, a standard request-response messaging exchange is not appropriate. Similarly, when a given consumer request requires that the service perform prolonged processing before being able to respond, synchronous communication is not possible without jeopardizing scalability and reliability of the service and its surrounding architecture (Figure 18.8).

**Figure 18.18**

Service A, acting as the service consumer, issues a request message to Service B (1), and because it is part of a synchronous data exchange, Service A is required to wait (2) until Service B processes the request message and then transmits a response (3). During this waiting period, both service and consumer must be available and continue to use up memory.

Solution

Services are designed in such a manner that consumers provide them with a callback address at which they can be contacted by the service at some point after the service receives the initial consumer request message (Figure 18.19). Consumers are furthermore asked to supply correlation details that allow the service to send an identifier within future messages so that consumers can associate them with the original task.

NOTE

A callback address does not need to represent the address of the consumer that provided it. The callback address can point to a different location altogether.

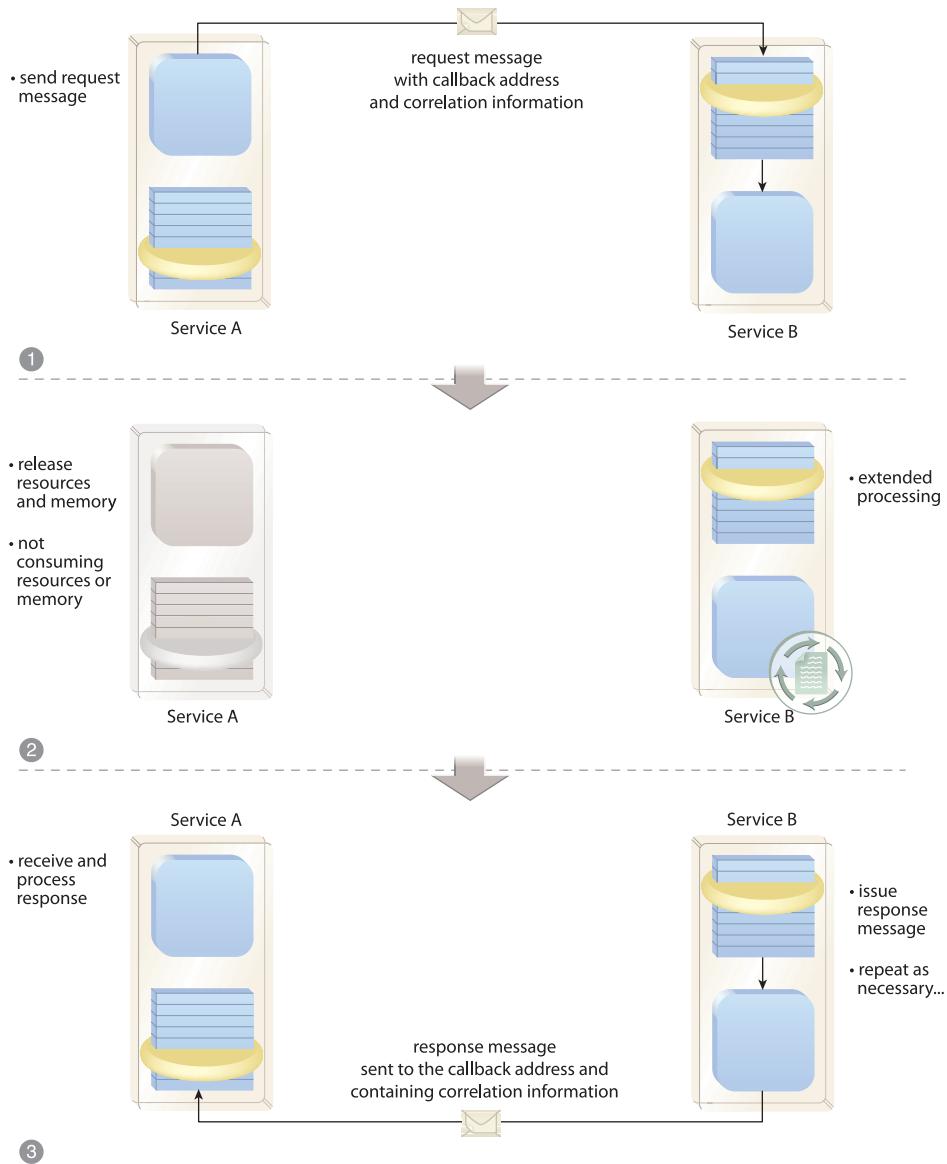
Application

Services designed to support this pattern must be able to preserve callback addresses and associated correlation data, especially when using this technique for longer running service-side processes that may include extended periods of inactivity (such as when waiting for human interaction to occur). While this may be built into the actual service architecture, the management of this information is often assumed by the surrounding inventory infrastructure, especially when implementing this pattern with established standards, such as WS-Addressing.

On the consumer side, this pattern can be supported through the use of event-driven agent programs that are positioned to listen for service response messages. This again may be provided by the infrastructure extensions themselves.

Furthermore, because a consumer will not be expecting an immediate response to its original request, it will commonly move on to other tasks while the service continues with its processing. The consumer architecture may therefore need to be able to support the scenario where service requests are received and temporarily stored until the consumer is ready to process them. This is why this pattern is often implemented with the support of messaging queues.

Also, because the callback address does the job of redirecting service responses, the service needs to ensure that the callback address is a trusted destination for its messages.

**Figure 18.19**

Service A sends a message containing the callback address and correlation information to Service B (1). While Service B is processing the message, Service A is unblocked (2). Service B, at some later point in time, sends a response containing the correlation information to the callback address to Service A (3). While Service B retains this callback address, it can continue to issue subsequent response messages to Service A.

Impacts

The asynchronous nature of the messaging introduced by this pattern can reduce reliability due to the absence of the immediate feedback received with standard synchronous exchanges. Reliable Messaging (592) can be applied to alleviate this risk.

Establishing an architecture whereby all services within a given inventory support WS-Addressing can introduce significant costs associated with necessary infrastructure upgrades. However, not all services will likely require this pattern, which may allow for its application to be limited to select composition architectures.

Relationships

Service Callback is based on the use of Service Messaging (533) and will normally require the application of Messaging Metadata (538) to represent the callback and correlation details separately from the message body content. In fact, Messaging Metadata (538) can be considered a critical requirement with respect to transferring the callback address and correlation information.

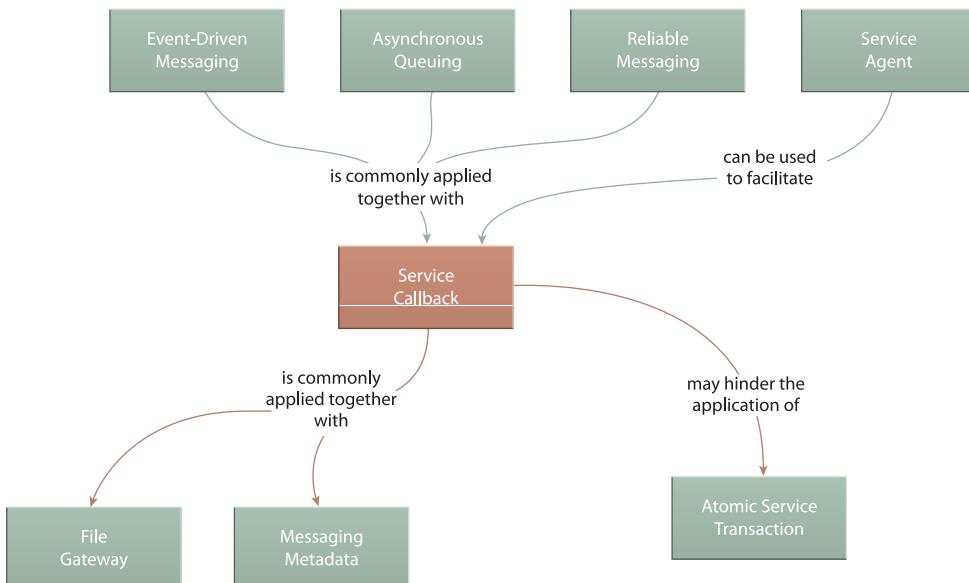


Figure 18.20

The decoupled nature of Service Callback naturally leads to relationships with other messaging patterns.

Furthermore, the need to coordinate asynchronous requests and possible multiple responses will typically involve Service Agent (543), especially when this pattern is applied as a result of infrastructure extensions.

While Service Callback may be considered an alternative to Asynchronous Queuing (582), it is common to use message queues together with issuing messages that contain callback addresses. This pattern can be supported through the use of Event-Driven Messaging (599) and Reliable Messaging (592).

NOTE

Service Callback incorporates concepts established in prior messaging patterns, including Return Address (Hohpe, Woolf) and Correlation Identifier (Hohpe, Woolf).

CASE STUDY EXAMPLE

Cutit Saw's recently released diamond blade chain design has become a runaway success, so much so that Cutit cannot keep up with the demand surge. This has not only resulted in large time gaps in order fulfillment, but also in higher costs as Cutit's IT department has had to track backorders manually. As a result, Cutit's response time has slowed dramatically. This, in turn, has raised concerns with Alleywood, one of its primary clients.

Cutit currently responds to purchase orders synchronously, and when backorders are required, the original purchase order is simply rejected. The manual process that was put in place before the success of the new blade design required Cutit to monitor their inventory and inform Alleywood about availability, as well as ask them to resubmit purchase orders after the items are back in stock. This not only scales poorly, but McPherson's new CIO does not like the cost that this process imposes on Alleywood.

Subsequent to a formal complaint lodged by the CIO in which McPherson threatens to equip all Alleywood workers with chainsaw blades from a competitor, Cutit's management scrambles to make changes. Senior architects are sent to the Alleywood office to meet with IT delegates in an attempt to work out a better system. The end-result is an agreement that Cutit will improve its response time by building a dedicated Backorder service capable of automating the backorder process asynchronously.

In support of this, Alleywood agrees that its consumer programs will provide callback addresses with each purchase order message sent to the Cutit Backorder service.

Specifically, they are required to provide the WS-Addressing `wsa:From` header block with each of their SOAP request messages. This header block specifies the callback address where asynchronous responses will need to be sent by the Backorder service.

If all the items in the purchase order are in stock, this service responds (to the callback address) asynchronously with an order fulfillment statement containing expected delivery date and shipping information. As backordered items become available, the service sends updates to this same callback address with new delivery date and shipping information until the purchase order is finally fulfilled.

In addition to the use of the `wsa:From` header block, the `wsa:MessageID` header is incorporated to represent correlation identifiers. Also the `wsa:RelatesTo` header is added with a special `RelationshipType` value of “<http://cutitsaws.com/po/response>” and the message ID of the request message. This information is used by Alleywood to correlate responses from Cutit to the original purchase order request.

The following code sample shows a purchase order request message from Alleywood to Cutit:

```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <Header>
    <wsa:From>
      <wsa:Address>
        http://mcpherson/alleywood/cutitPO
      </wsa:Address>
    </wsa:From>
    <wsa:MessageID>
      http://mcpherson/6B29FC40-CA47-1067-B31D-00DD010662DA
    </wsa:MessageID>
    <wsa:Action>
      ...
    </wsa:Action>
  </Header>
  <Body>
    <!-- PO details -->
    ...
  </Body>
</Envelope>
```

Example 18.5

A purchase order request message that uses the `wsa:From` construct to specify the callback address where Cutit can send one or more responses asynchronously.

This next example shows the subsequent response message issued by Cutit back to Alleywood's callback address:

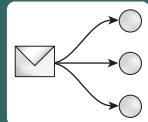
```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <Header>
    <wsa:To>
      http://mcpherson/alleywood/cutitPO
    </wsa:To>
    <wsa:RelatesTo RelationshipType=
      "http://cutitsaws.com/po/response">
      http://mcpherson/6B29FC40-CA47-1067-B31D-00DD010662DA
    </wsa:RelatesTo>
    <wsa:Action>
      ...
    </wsa:Action>
  </Header>
  <Body>
    <!-- PO response from Cutit -->
    ...
  </Body>
</Envelope>
```

Example 18.6

This response message is sent to the location specified in the `wsa:From` header from the corresponding request message. It uses the `wsa:MessageID` element from that request message as its correlation identifier. Note also how the value of the `wsa:RelatesTo` construct matches that of the `wsa:MessageID` construct from Example 18.5.

Service Instance Routing

By Anish Karmarkar



How can consumers contact and interact with service instances without the need for proprietary processing logic?

Problem	When required to repeatedly access a specific stateful service instance, consumers must rely on custom logic that more tightly couples them to the service.
Solution	The service provides an instance identifier along with its destination information in a standardized format that shields the consumer from having to resort to custom logic.
Application	The service is still required to provide custom logic to generate and manage instance identifiers, and both service and consumer require a common messaging infrastructure.
Impacts	This pattern can introduce the need for significant infrastructure upgrades and when misused can further lead to overly stateful messaging activities that can violate the Service Statelessness principle.
Principles	Service Loose Coupling, Service Statelessness, Service Composability
Architecture	Inventory, Composition, Service

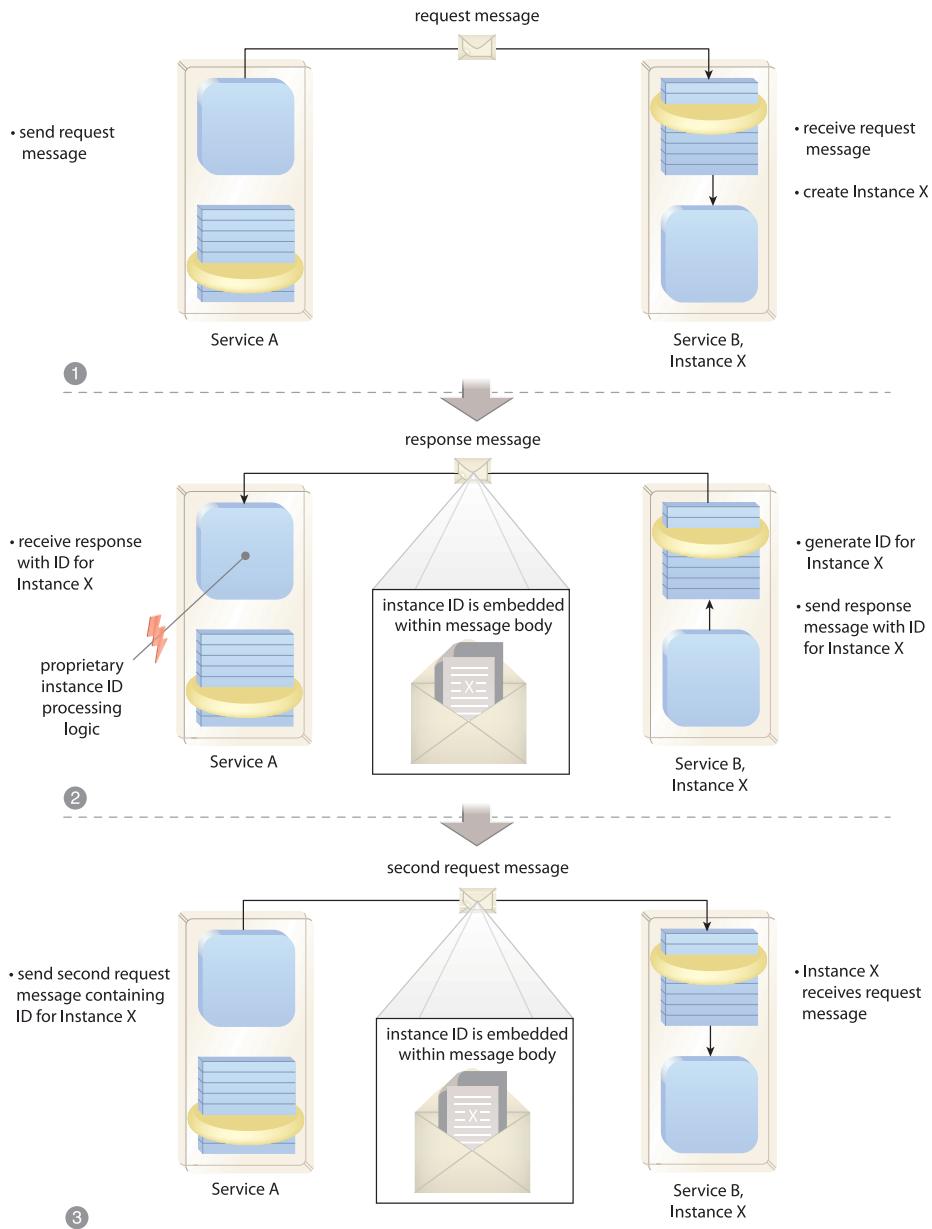
Table 18.7

Profile summary for the Service Instance Routing pattern.

Problem

There are cases where a consumer sends multiple messages to a service and the messages need to be processed within the same runtime context. Such services are intentionally designed to remain stateful so that they can carry out conversational or session-centric message exchanges.

However, service contracts generally do not provide a standardized means of representing or targeting instances of services. Therefore, consumer and service designers need to resort to passing proprietary instance identifiers as part of the regular message data, which results in the need for proprietary instance processing logic (Figure 18.21).

**Figure 18.21**

Service A, acting as a service consumer, issues a request message to Service B. An instance of Service B is created (1) using proprietary internal service logic that labels the instance as "Instance X." Service B returns an identifier for Instance X as part of the response message body back to Service A (2). Proprietary processing logic within Service A locates and extracts the embedded instance identifier and then embeds it into a second message that it sends to Instance X of Service B (3).

Note that in the scenario depicted in Figure 18.21, the lifecycle of the instances and the routing of the messages are managed by Service B. Throughout this exchange, Service A remains aware of any instance identifiers generated by Service B. Given that every such conversation can be different, there is no uniformity, and instance details are always required to be processed by custom logic that ends up increasing the coupling between the service and any of its consumers.

Solution

The underlying infrastructure is extended to support the processing of message metadata that enables a service instance identifier to be placed into a reference to the overall destination of the service (Figure 18.22). This reference (also referred to as an *endpoint reference*) is managed by the messaging infrastructure so that messages issued by the consumer are automatically routed to the destination represented by the reference.

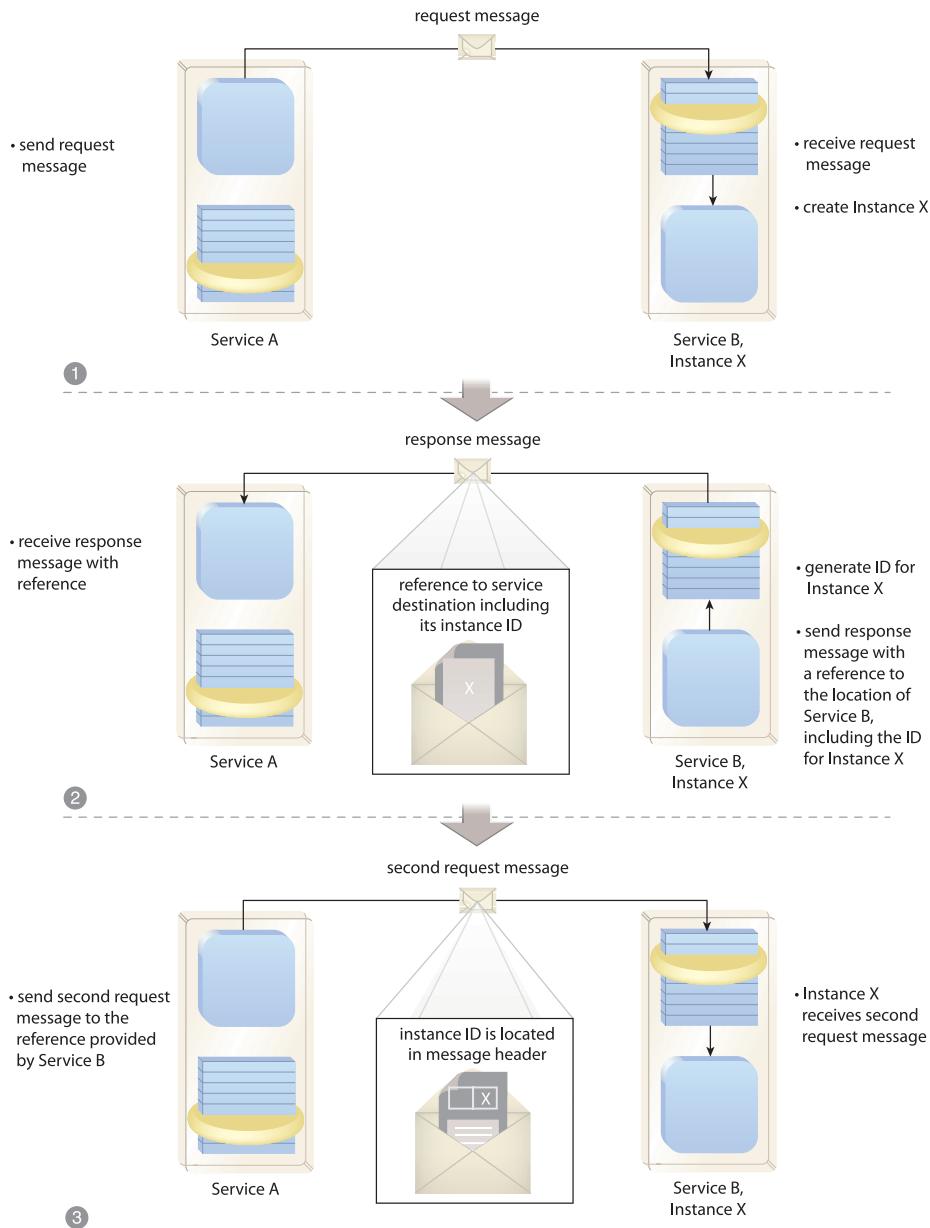
As a result, the processing of instance IDs does not negatively affect consumer-to-service coupling because consumers are not required to contain proprietary service instance processing logic. Because the instance identifiers are part of a reference that is managed by the infrastructure, they are opaque to consumers. This means that consumers do not need to be aware of whether they are sending messages to a service or one of its instances because this is the responsibility of the routing logic within the messaging infrastructure.

Application

The echoing of the service instance identifier in conversational messages needs to be incorporated on the consumer-side messaging framework and architecture. An infrastructure that supports service instance routing is therefore required. When building services as Web services, this pattern is typically applied using infrastructure extensions compliant with the WS-Addressing specification.

NOTE

This pattern also forms the basis for providing additional *service-side* support for conversations or sessions. In such cases, the lifecycle management of instances is delegated to the infrastructure and codified by container contracts available via application server containers.

**Figure 18.22**

Service A, acting as a service consumer, issues a request message to Service B. Instance X of Service B is created (1), and a new message containing a reference to the destination of Service B (which includes the Instance X identifier) is returned back to Service A (2). Service A issues a second message that is routed to Instance X of Service B (3) without the need for proprietary logic. The instance identifier is located in the header of this message and is therefore kept separate from the message body.

Impacts

Applying this pattern across an entire service inventory requires that the necessary infrastructure extensions be established as part of the inventory architecture. This can lead to increased costs and governance effort.

Service Instance Routing can be used to create highly stateful services designed to carry out prolonged conversational message exchanges. While stateful interaction is often required, it is easy to apply this pattern to such an extent that it runs contrary to the Service Statelessness principle, thereby undermining the importance of long-term service scalability.

Furthermore, because service instance identifiers are valid only during the lifecycle of the instance, there is the danger that stale identifiers may be inadvertently used for invocation. Controls are required to ensure that identifiers are destroyed after the end of each service instance.

Relationships

Service Instance Routing is naturally related to Messaging Metadata (538) because it applies to messages that generally require the use of message headers. The actual mechanics behind the implementation of the infrastructure extensions necessary for this pattern often rely on the use of event-driven intermediaries to carry out the message header process, which is why this pattern is frequently associated with Service Agent (543).

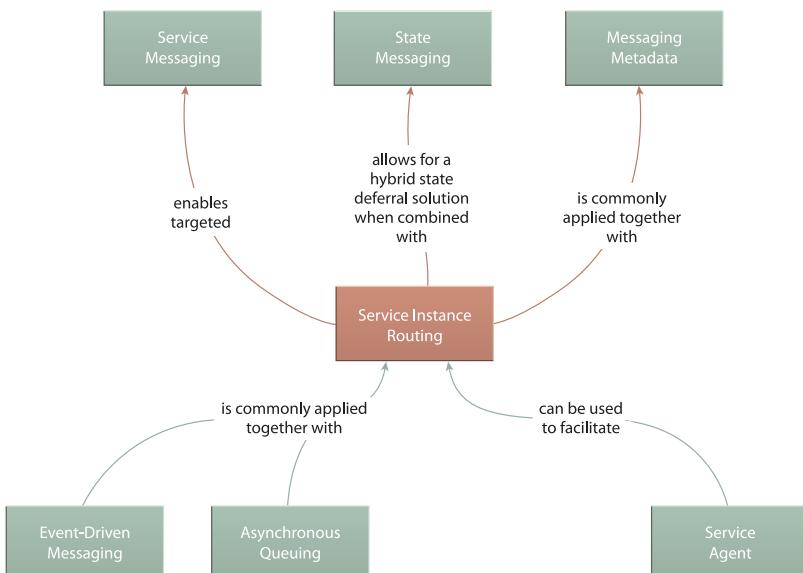


Figure 18.23

The modification of service reference structure and echoing of instance identifiers as opaque tokens affects several other messaging patterns.

CASE STUDY EXAMPLE

Based on a recent marketing research report, management at Cutit Saws has decided to make their products available to the public. To accomplish this, Cutit architects are asked to establish an online presence with a standard Web site that contains a shopping cart that customers can use to pick and choose products for purchase and delivery.

The new External Order Web service is created as the backend engine for the new Web site. This service is contacted each time the online shopping cart is used. Once invoked, its instance remains in memory, and the service is considered stateful.

One instance of the External Order service instance corresponds to one online shopping cart. The Web site scripts need to obtain some sort of identifier to the External Order service instance so that they can continue interacting with it while the human end-user manages the shopping cart.

After reviewing these very specific requirements, architects choose to invest in an infrastructure upgrade that introduces support for WS-Addressing Endpoint References (EPRs). EPRs essentially provide them with a standardized means of expressing the address of a Web service along with a set of parameters (called reference parameters) that can be used to specify the service instance.

The following excerpt highlights a fragment of the Cutit implementation of WS-Addressing EPRs by showing the contents of the Create Shopping Cart request message:

```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope">
  <Header>
    ...
  </Header>
  <Body>
    <cu:CreateShoppingCart
      xmlns:cu="http://cutitsaws.com/shopping">
      ...
    </cu:CreateShoppingCart>
  </Body>
</Envelope>
```

Example 18.7

A request message for creating a new shopping cart.

The next example displays the corresponding response message issued from the External Order service, which returns the WS-Addressing EPR cu:ShoppingCartEPR, as follows:

```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <Header>
    ...
  </Header>
  <Body>
    <!-- Response from Cutit -->
    <cu:CreateShoppingCartResponse
      xmlns:cu="http://cutitsaws.com/shopping">
      <cu:ShoppingCartEPR>
        <wsa:Address>
          http://cutitsaws.com/shoppingcart
        </wsa:Address>
        <wsa:ReferenceParameters>
          <cu:InstanceID>
            6B29FC40-CA47-1067-B31D-00DD010662DA
          </cu:InstanceID>
        </wsa:ReferenceParameters>
      </cu:ShoppingCartEPR>
    </cu:CreateShoppingCartResponse>
    ...
  </Body>
</Envelope>
```

Example 18.8

Note how this message contains a reference parameter called cu:InstanceID in addition to the standard network endpoint details. This reference parameter contains the identifier of the service instance that the consumer is expected to use for subsequent requests.

This final example shows the contents of the message sent back to the External Order service instance:

```
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://www.w3.org/2005/08/addressing"
  xmlns:cu="http://cutitsaws.com/shopping">
  <Header>
    <wsa:To>
      http://cutitsaws.com/shoppingcart
    </wsa:To>
    <cu:InstanceID wsa:IsReferenceParameter="true">
      6B29FC40-CA47-1067-B31D-00DD010662DA
    </cu:InstanceID>
```

```
...
</Header>
<Body>
    <!-- Message sent to a particular shopping cart -->
    <cu:AddItem>
        ...
    </cu:AddItem>
</Body>
</Envelope>
```

Example 18.9

This cu:AddItem message is targeted to a specific instance of the Cutit External Order service.

Asynchronous Queuing

By Mark Little, Thomas Rischbeck, Arnaud Simon



How can a service and its consumers accommodate isolated failures and avoid unnecessarily locking resources?

Problem	When a service capability requires that consumers interact with it synchronously, it can inhibit performance and compromise reliability.
Solution	A service can exchange messages with its consumers via an intermediary buffer, allowing service and consumers to process messages independently by remaining temporally decoupled.
Application	Queuing technology needs to be incorporated into the surrounding architecture, and back-up stores may also be required.
Impacts	There may be no acknowledgement of successful message delivery, and atomic transactions may not be possible.
Principles	Standardized Service Contracts, Service Loose Coupling, Service Statelessness
Architecture	Inventory, Composition

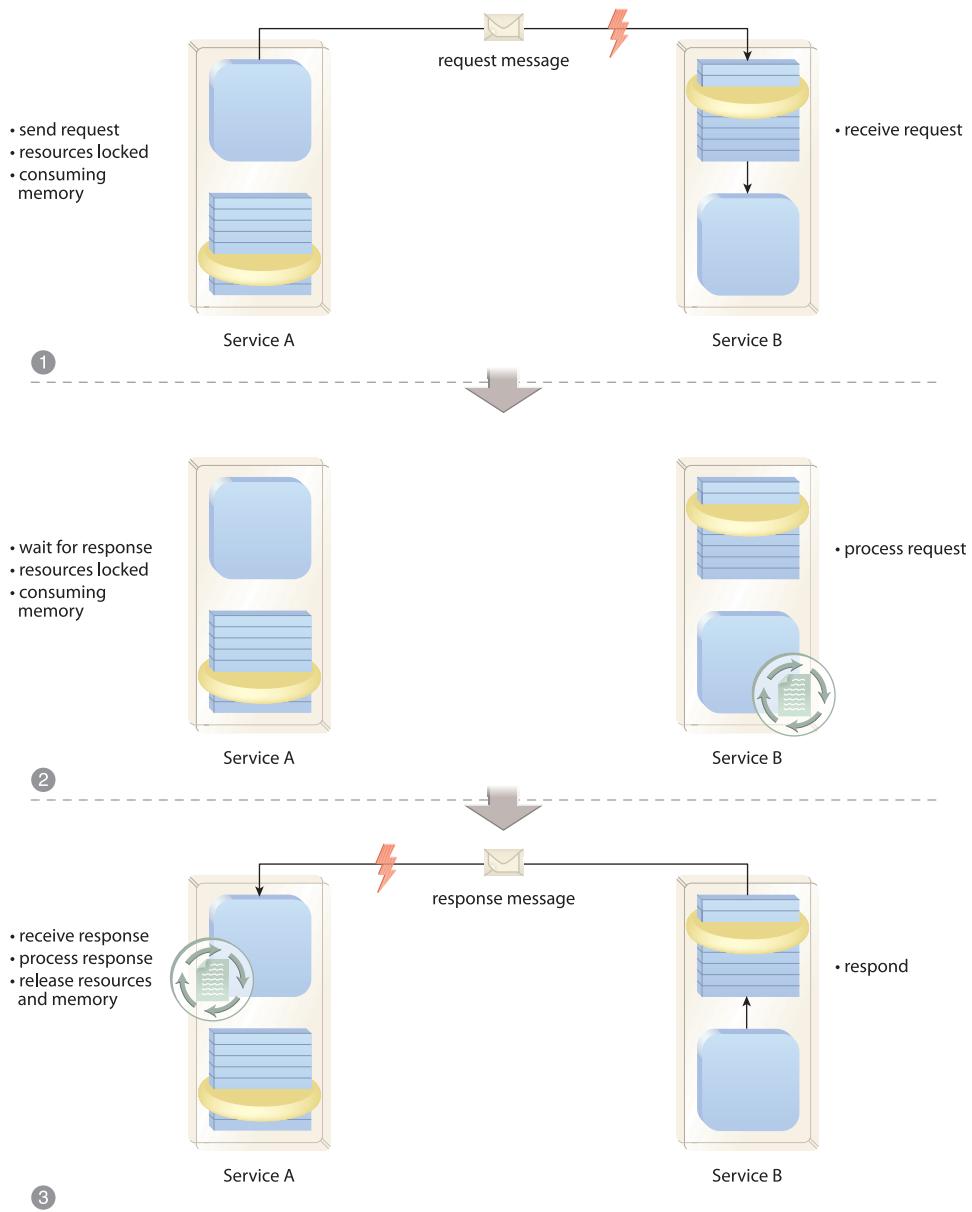
Table 18.8

Profile summary for the Asynchronous Queuing pattern.

Problem

Synchronous communication requires an immediate response to each request and therefore forces two-way data exchange for every service interaction. When services need to carry out synchronous communication, both service and service consumer must be available and ready to complete the data exchange. This can introduce reliability issues when either the service cannot guarantee its availability to receive the request message or the service consumer cannot guarantee its availability to receive the response to its request.

Because of its sequential nature, synchronous message exchanges can further impose processing overhead, as the service consumer needs to wait until it receives a response from its original request before proceeding to its next action. As shown in Figure 18.24, prolonged responses can introduce latency by temporally locking both consumer and service.

**Figure 18.24**

Service A, acting as the service consumer, issues a request message to Service B (1), and because it is part of a synchronous data exchange, Service A is required to wait (2) until Service B processes the request message and then transmits a response (3). During this waiting period, both service and consumer must be available and continue to use up memory. Because Asynchronous Queueing and Service Callback (566) both enable asynchronous messaging as an alternative to synchronous communication, this figure is identical to Figure 18.18, except for the red lightning bolt symbols which hint at the reliability problem also addressed by this pattern.

Another problem forced synchronous communication can cause is an overload of services required to facilitate a great deal of concurrent access. Because services are expected to process requests as soon as they are received, usage thresholds can be more easily reached, thereby exposing the service to multi-consumer latency or overall failure.

Solution

A queue is introduced as an intermediary buffer that receives request messages and then forwards them on behalf of the service consumers (Figure 18.25). If the target service is unavailable, the queue acts as temporary storage and retains the message. It then periodically attempts retransmission.

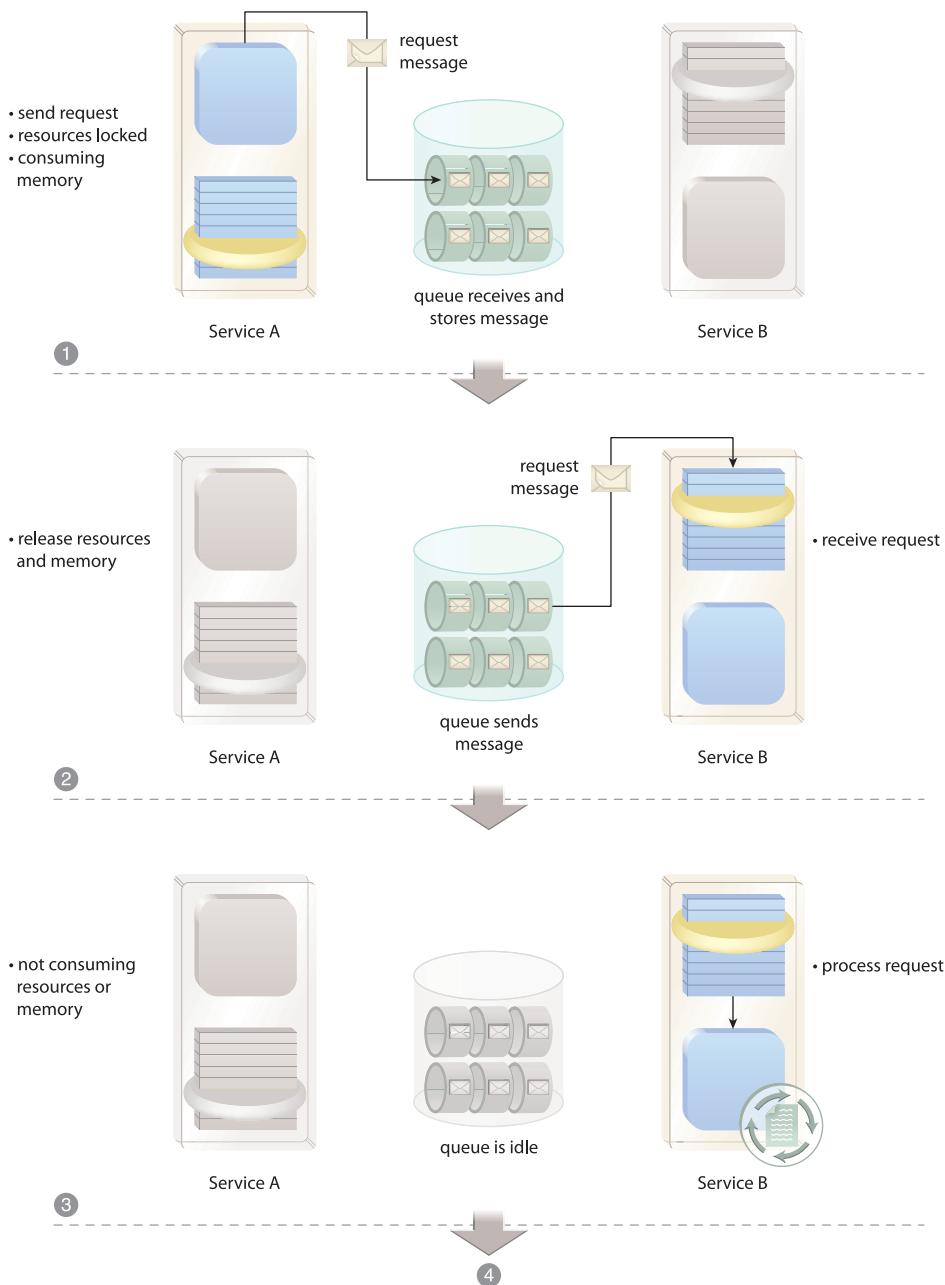
Similarly, if there is a response, it can be issued through the same queue that will forward it back to the service consumer when the consumer is available. While either service or consumer is processing message contents, the other can deactivate itself (or move on to other processing) in order to minimize memory consumption (Figure 18.26).

Application

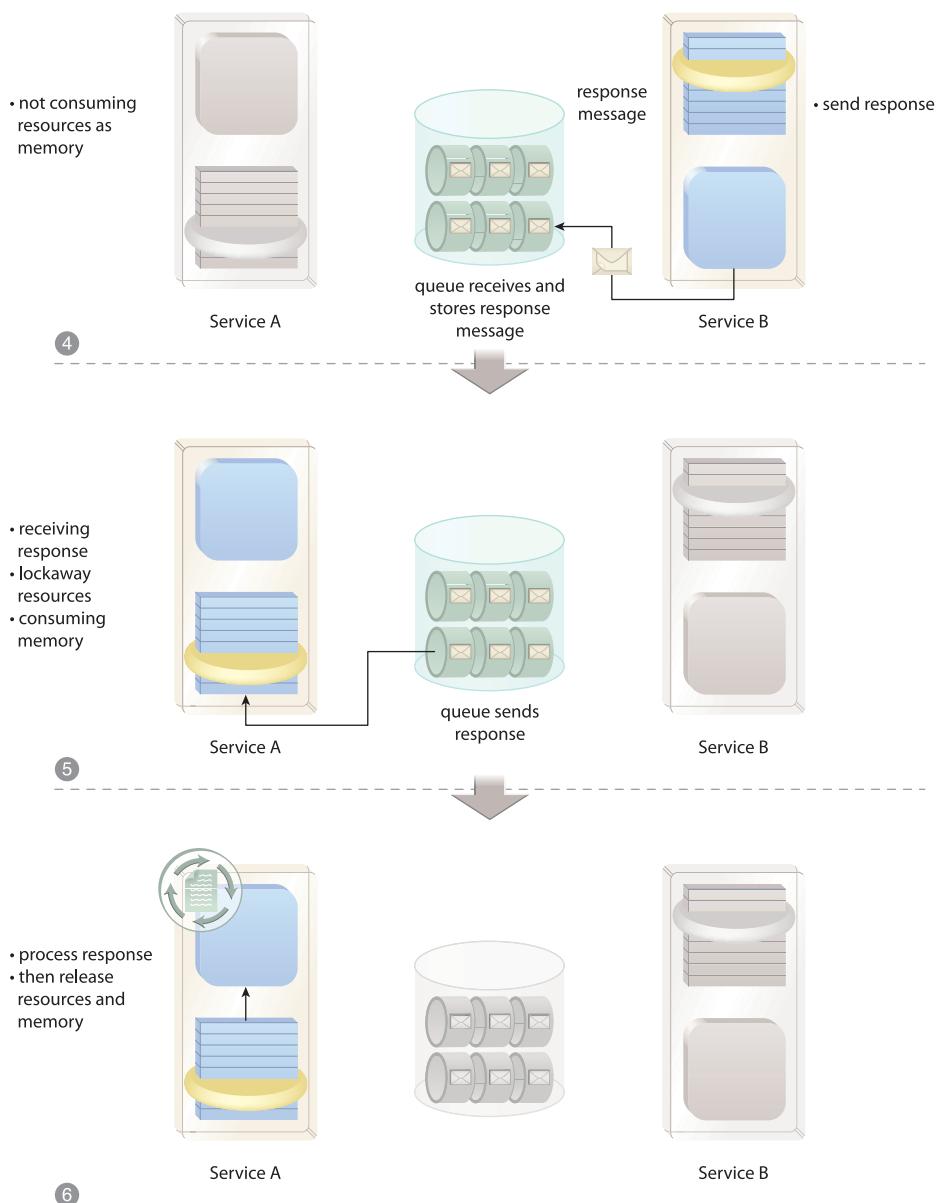
In modern ESB platforms, the use of a queue can be completely transparent, meaning that neither consumer nor service may know that a queue was involved in a data exchange. The queuing framework can be supported by intelligent service agents that detect when a queue is required and intercept message transmissions accordingly.

The queue can be configured to process messages in different ways and is typically set up to poll an unavailable target recipient periodically until it becomes available or until the message transmission is considered to have failed. Queues can further be used to leverage asynchronous message exchange by incorporating topics and message broadcasts to multiple consumers, as per Event-Driven Messaging (599).

Many vendor queues are equipped with a back-up store so that messages in transit are not lost should a system failure occur. Especially when supporting more complex compositions, Asynchronous Queuing is commonly applied in conjunction with Reliable Messaging (592).

**Figure 18.25**

Service A sends a message to Service B, which is intercepted and stored by an intermediary queue (1). The queue forwards the message to Service B (2), and while Service B is processing the message, Service A remains released from memory (3).

**Figure 18.26**

After completing its processing, Service B issues a response message back to Service A, which is also received and stored by the intermediary queue (4). Service A receives the response (5) and completes processing of the response, all the while Service B is deactivated (6).

NOTE

In some platforms, services to which the queue is expected to forward messages need to be pre-registered with the queue in advance. There are other common characteristics about the use of messaging queues that are not explained in this pattern, such as a “pull” based architecture wherein services are required to poll the queue to retrieve messages instead of the “push” model described so far. This pattern does not intend to describe the usage of messaging queues in general; it is focused solely on asynchronous messaging in response to the problem defined in the preceding *Problem* section.

Impacts

The use of intermediary queues allows for creative asynchronous message exchange patterns that can optimize service interaction by eliminating the need for a required response to each request. However, asynchronous message exchanges can also lead to more complex service activities that are difficult to design. It may be challenging to anticipate all of the possible runtime scenarios at design-time, and therefore extra exception handling logic may be necessary.

An asynchronous data exchange that involves a queue can also be more difficult to control and monitor. It may not be possible to protect asynchronous activities with Atomic Service Transaction (623) because of the time-response constraints usually associated with transactions and their requirements to hold resources in suspension until either commit or roll-back instructions are issued.

Furthermore, an advantage to synchronous messaging is that because a response is always required, it acts as an immediate acknowledgement that the initial request message was successfully delivered and processed. With asynchronous message exchange patterns, no response is expected, and the message issuer therefore is not necessarily notified of successful or failed deliveries. However, most queuing systems allow the monitoring and administration of in-flight message transmissions. Messages in the queue can be further examined and managed during transit, which in larger systems can greatly simplify administrative control and the isolation of communication faults.

Relationships

Asynchronous Queuing is a design pattern dedicated to accommodating message exchanges and therefore is naturally related to Service Messaging (533). Event-driven agents form a fundamental part of the queuing framework, which explains the relevance of Service Agent (543). Furthermore, Messaging Metadata (538) can play a role in how messages are processed, stored, or routed via these agents.

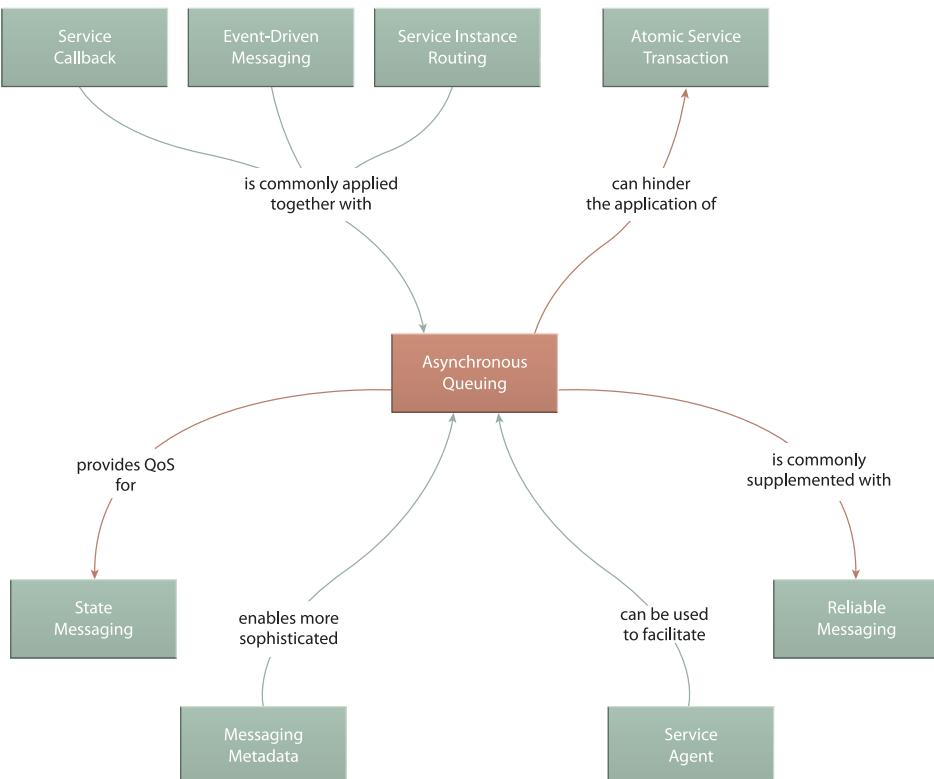


Figure 18.27

The message-centric focus of Asynchronous Queuing naturally leads to relationships with other messaging patterns.

ESB platforms are fundamentally about decreasing the coupling between different parts of a service-oriented solution, which is why this pattern is a core part of Enterprise Service Bus (704), as shown in Figure 18.28.

An optional design pattern associated with the ESB is Reliable Messaging (592), which is a pattern commonly applied in conjunction with Asynchronous Queuing. Together, these two patterns provide key QoS extensions that make the use of ESB products attractive, especially in support of complex service compositions.

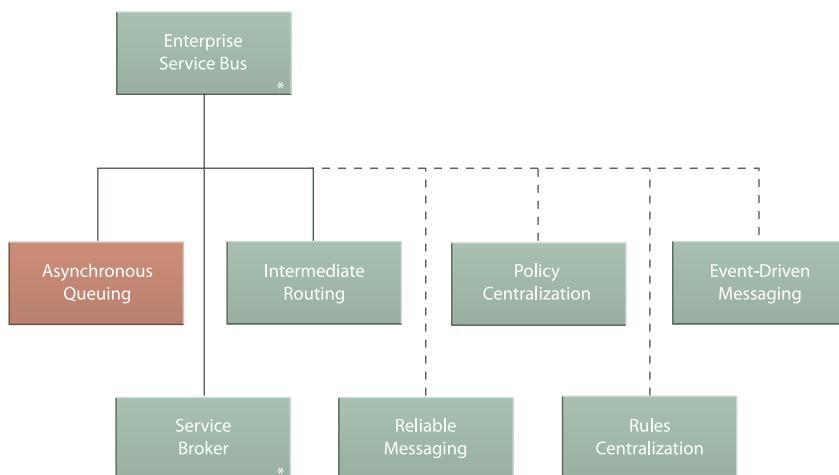


Figure 18.28

Asynchronous Queuing is a design pattern that can be applied independently but also represents one of the core patterns that comprise Enterprise Service Bus (704).

CASE STUDY EXAMPLE

The FRC continues to employ an integrated legacy environment for the transfer of batch extracts from an ERP system to a proprietary purchase order administration (POAdmin) product. The ERP environment provides an API capable of outputting exported data from its native CSV format to a predefined XML structure. An ERP wrapper Web service was further made available by the vendor in order to enable access to extracted data via SOAP.

Generating the extracts was time consuming, especially if purchase order data for a range of dates was requested. The batch files could also be very large, which required longer transfer times. As a result, there have been a variety of performance issues due to

the fact that the POAdmin program had to remain locked and suspended while waiting for the response from the ERP API, which required a synchronous exchange (Figure 18.29). This also caused the occasional system crash as the POAdmin request would timeout, resulting in the POAdmin program to freeze.

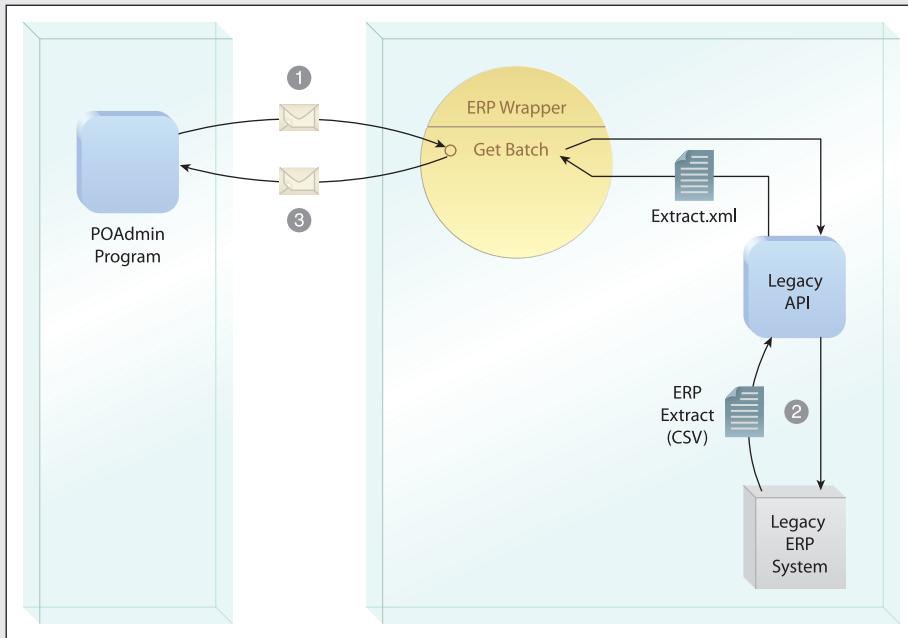
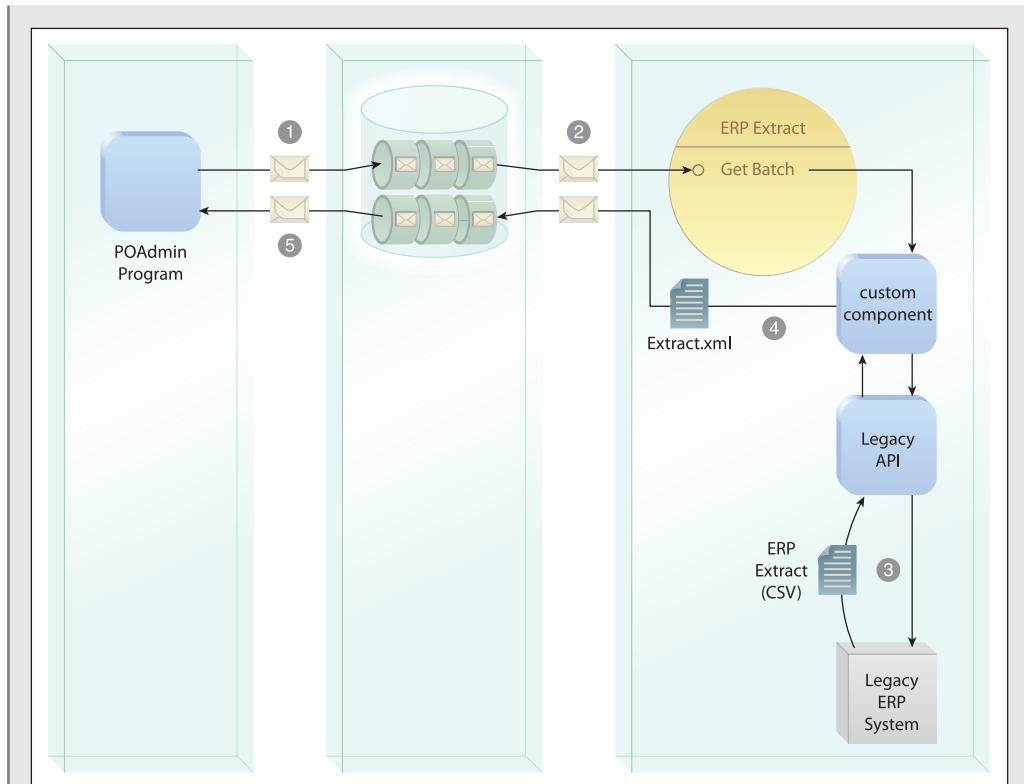


Figure 18.29

The legacy POAdmin program initiates a synchronous exchange by issuing a request message to the ERP Wrapper service (1). While the underlying ERP environment retrieves the requested batch extract (2), the POAdmin program continues to wait in suspension until it finally receives a response comprised of the extract in XML format (3).

Network outages have also occurred, leading to interruptions half-way through a transfer. In this situation, the whole transfer would need to be repeated. This required frequent administrative intervention, and management perceived the solution as brittle and inefficient.

As illustrated in Figure 18.30, the FRC project team decides to re-architect this environment to improve the reliability of this exchange but also to provide more appropriate access to the ERP environment for additional services that may need to work with its data.

**Figure 18.30**

The POAdmin program sends a request that is received by the queue (1). The queue then forwards the message to the new ERP Extract service (2), which then passes it along to a custom component. This component interacts with the ERP API to retrieve the requested extract data (3) and then forwards this directly to the queue (4). The queue then interacts with the POAdmin API to provide the program the data it originally requested.

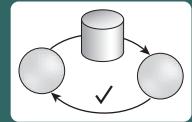
This new architecture introduces a queue to store and then forward all requests on behalf of the POAdmin program. The queue is configured to issue an immediate response to POAdmin in order to simulate the synchronous exchange. A new, standardized Web service is then brought in to replace the previous ERP Wrapper service. This new ERP Extract service provides a one-way operation that simply receives the request message from the queue. It then interacts with an (also newly provided) custom component that interfaces with the ERP API.

Once the extract is received, this component directly forwards the result to the queue, which in turn accesses the POAdmin API to transmit the purchase order data extract.

Reliable Messaging

By Mark Little, Thomas Rischbeck, Arnaud Simon

How can services communicate reliably when implemented in an unreliable environment?



Problem	Service communication cannot be guaranteed when using unreliable messaging protocols or when dependent on an otherwise unreliable environment.
Solution	An intermediate reliability mechanism is introduced into the inventory architecture, ensuring that message delivery is guaranteed.
Application	Middleware, service agents, and data stores are deployed to track message deliveries, manage the issuance of acknowledgements, and persist messages during failure conditions.
Impacts	Using a reliability framework adds processing overhead that can affect service activity performance. It also increases composition design complexity and may not be compatible with Atomic Service Transaction (623).
Principles	Service Composability
Architecture	Inventory, Composition

Table 18.9

Profile summary for the Reliable Messaging pattern.

Problem

When services are designed to communicate via messages, there is a natural loss of quality-of-service due to the stateless nature of underlying messaging protocols, such as HTTP. Unlike with binary communication protocols where a persistent connection is maintained until the data transmission between a sender and receiver is completed, with message exchanges the runtime platform may not be able to provide feedback to the sender as to whether or not a message was successfully delivered (Figure 18.31).

Furthermore, because the probability of failure is exacerbated as the service count (and the number of corresponding network links) grows with service compositions increasing in size and complexity, the inability of an infrastructure to introduce guaranteed message delivery can introduce measurable risk factors into service composition architectures (especially those that rely heavily on agnostic services).

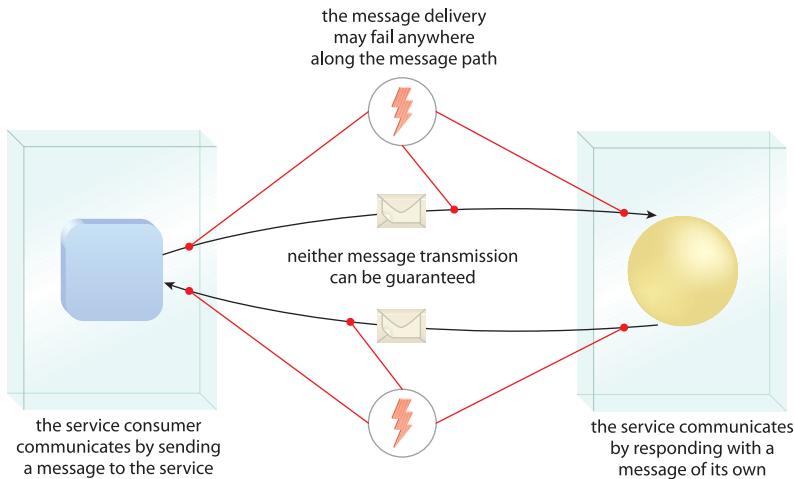


Figure 18.31

During the course of a regular message exchange, there are no guarantees. Various runtime conditions may cause the message delivery to fail.

Solution

The inventory architecture is equipped with a reliability framework that tracks and temporarily persists message transmissions and issues positive and negative acknowledgements to communicate successful and failed transmissions to message senders.

Application

A complete reliability framework is typically comprised of infrastructure and intermediary processing logic capable of:

- guaranteeing message delivery during failure conditions via the use of a persistence store
- tracking messages at runtime
- issuing acknowledgements for individual or sequences of messages

The repository used for guaranteed delivery may provide the option to store messages in memory or on disk so as to act as a back-up mechanism for when message transmissions fail. This central storage also eases the management and administration of service-oriented solutions because it allows administrators to track the status of messages and trace the causes behind unresolved delivery problems.

Reliability agents further manage the confirmation of successful and failed message deliveries via positive (ACK) and negative (NACK) acknowledgement notifications. Messages may be transmitted and acknowledged individually, or they may be bundled into message sequences that are acknowledged in groups (and may also have sequence-related delivery rules).

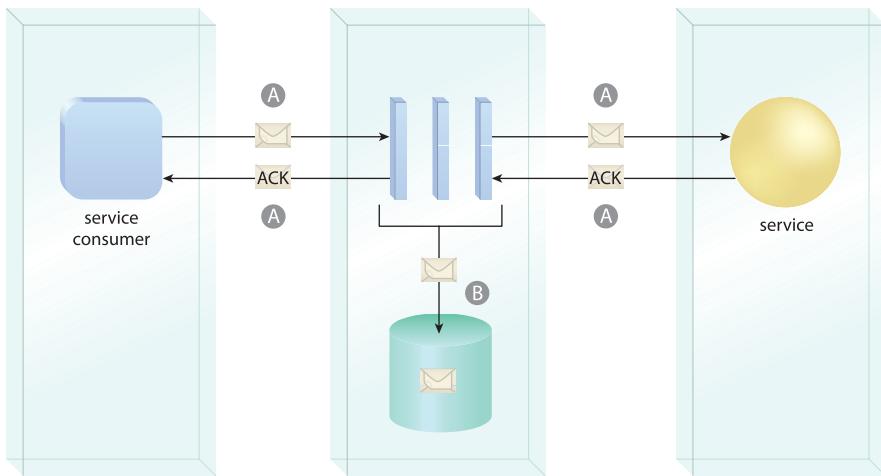


Figure 18.32

When building services as Web services, this pattern is commonly applied by implementing a combination of the WS-ReliableMessaging standard (A) and guaranteed delivery extensions, such as a persistent repository (B). This figure highlights the typical moving parts of the resulting reliability framework.

Impacts

Reliable Messaging introduces a layer of processing that includes runtime message capture, persistence, tracking, and acknowledgement notification issuance. All of these features add moving parts to an inventory architecture that demand additional performance and guarantee requirements and increase the complexity of service-oriented solutions proportional to the size of their service compositions.

Furthermore, due to the temporary storage of messages, the incorporation of positive and negative acknowledgement notifications, and the use of various delivery rules (including those based on group message delivery via sequences), it may not be possible to wrap services using reliability features into atomic transactions, as per Atomic Service Transaction (623).

Relationships

Applying this pattern directly affects messaging-related patterns in that it changes how messages are transmitted and delivered. The quality of Service Messaging (533) is improved, and Messaging Metadata (538) is commonly utilized to manage and track messages via reliability agents that can be considered specialized implementations of Service Agent (543). Considerations arise from the application of Canonical Resources (237) can help ensure that an inventory architecture standardizes on a single reliability framework .

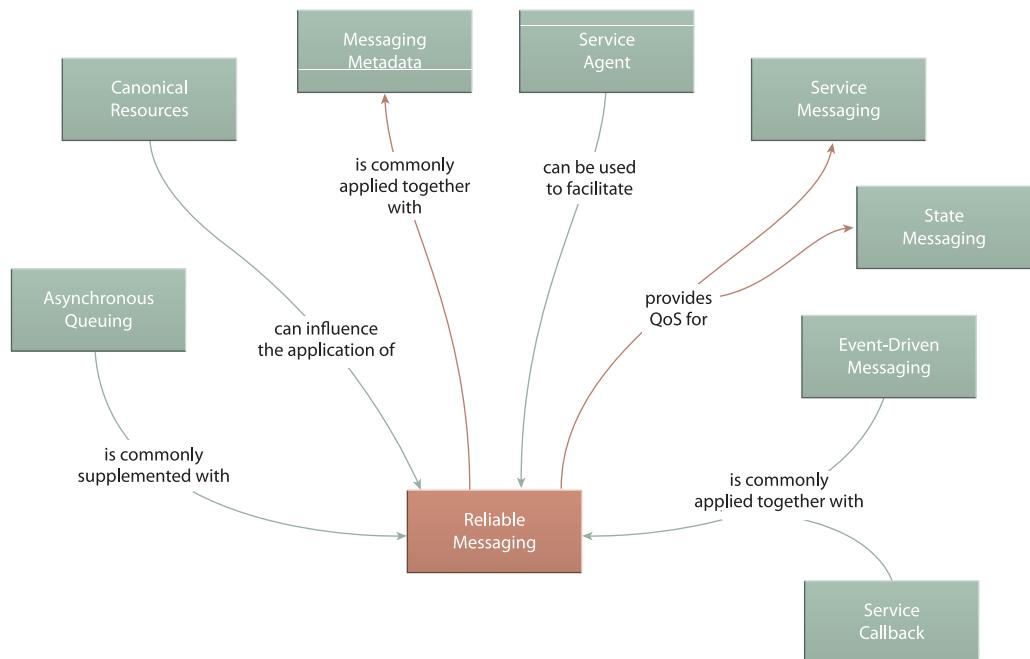


Figure 18.33

The messaging-centric focus of this pattern makes it naturally affect other messaging-related patterns.

Because of the importance of guaranteeing message delivery and improving the overall quality of base messaging frameworks, the runtime functionality established by applying Reliable Messaging is typically associated with ESB platforms (Figure 18.34).

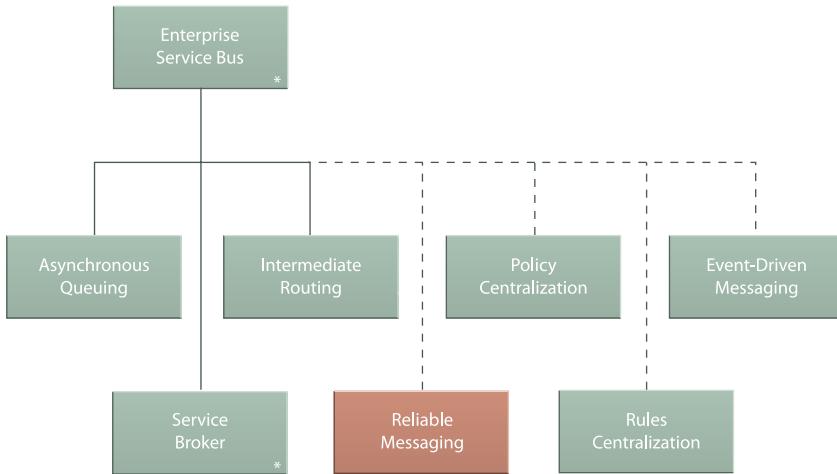


Figure 18.34

Reliable Messaging can be applied by itself, but is also a pattern commonly realized via Enterprise Service Bus (704).

NOTE

Reliable Messaging encompasses much of what Guaranteed Delivery (Hohpe, Woolf) advocates in relation to persisting messages and then further supplements this with additional reliability extensions.

CASE STUDY EXAMPLE

The FRC receives a great number of documents from external companies. Every day, in fact, thousands of messages are received via existing Web service endpoints, containing applications, requests, appeals, and even payment details.

Each month statistics are collected by IT administrators that show the quantity of messages received along with metrics associated with the number of failed deliveries, rejected deliveries, messages tagged as security threats, and general delivery errors. One statistic that has always been closely watched is “lost messages.” This represents messages that were received by the Web service but then went missing once forwarded to additional internal services.

The FRC has only been using Web services as their primary access point for external communication for the past six months. During that time, the lost message count has been consistently high. It was originally thought that the Web services platform itself required further tuning, but now, a half year later, it is considered a persistent problem that is simply unacceptable.

A new project is started specifically to address this and other quality of service issues with the externally facing Web services. Subsequent to an internal analysis and an assessment of the marketplace, the FRC architecture team decides to bring in an ESB product that provides built-in reliable messaging. Web services are redeployed within this new environment, and additional development and configuration effort is performed to incorporate the automatic acknowledgement system provided by the reliable messaging extensions.

This new architecture helps attain increased quality of service by reducing lost messages. The approach is to first group messages by type and then send each group to a corresponding internal service responsible for processing a type of message.

Each group of messages transmitted to an internal service is received with a positive or negative acknowledgement that communicates the success or failure of the deliveries (Figure 18.35). Upon receiving a negative acknowledgement, the Web service logs the failure and all of the message details. This establishes a permanent record of all failed delivery attempts.

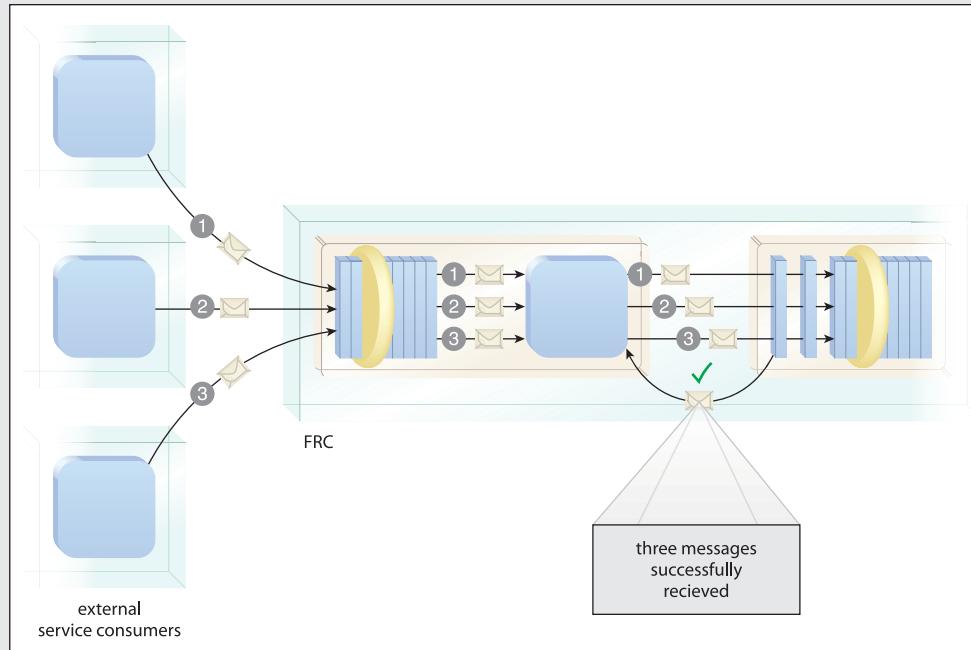
As shown in the following example, the use of reliable messaging headers allows the FRC to assign an identifier to the message along with a message number that indicates the position of the current message within the overall sequence group.

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"  
    xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"  
    xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">  
    <Header>  
        <wsrm:Sequence>  
            <wsu:Identifier>  
                uuid:038857-524  
            </wsu:Identifier>  
            <wsrm:MessageNumber>  
                9  
            </wsrm:MessageNumber>  
        </wsrm:Sequence>
```

```
</Header>
<Body>
...
</Body>
</Envelope>
```

Example 18.10

Through the use of reliable messaging metadata headers, this message is assigned a number that indicates where the message is located within the overall message sequence.

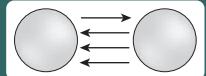
**Figure 18.35**

The externally facing FRC Web service receives a range of messages (1, 2, 3) from companies that have automated their data exchange with the FRC. With the reliable messaging extensions in place, messages are first grouped into sequences and then sent to the appropriate internal service for further processing.

Although the team is pleased with the manner in which they have instilled their inventory architecture with support for WS-ReliableMessaging, they acknowledge that this represents only a partial application of this pattern due to the absence of an intermediate persistence repository.

Event-Driven Messaging

By Mark Little, Thomas Rischbeck, Arnaud Simon



How can service consumers be automatically notified of runtime service events?

Problem	Events that occur within the functional boundary encapsulated by a service may be of relevance to service consumers, but without resorting to inefficient polling-based interaction, the consumer has no way of learning about these events.
Solution	The consumer establishes itself as a subscriber of the service. The service, in turn, automatically issues notifications of relevant events to this and any of its subscribers.
Application	A messaging framework is implemented capable of supporting the publish-and-subscribe MEP and associated complex event processing and tracking.
Impacts	Event-driven message exchanges cannot easily be incorporated as part of Atomic Service Transaction (623), and publisher/subscriber availability issues can arise.
Principles	Standardized Service Contract, Service Loose Coupling, Service Autonomy
Architecture	Inventory, Composition

Table 18.10

Profile summary for the Event-Driven Messaging pattern.

Problem

In typical messaging environments, service consumers can choose between one-way and request-response message exchange patterns (MEPs), but both need to originate from the consumer. Events may occur within the service provider's functional boundary that are of interest to the consumer. Following traditional MEPs, the consumer would need to continually poll the service in order to find out whether such an event had occurred (and to then retrieve the corresponding event details).

This model is inefficient because it leads to numerous unnecessary service invocations and data exchanges (Figure 18.36). It can further introduce delays as to when the consumer receives the event information because it may be only able to check for the event at predetermined polling intervals.

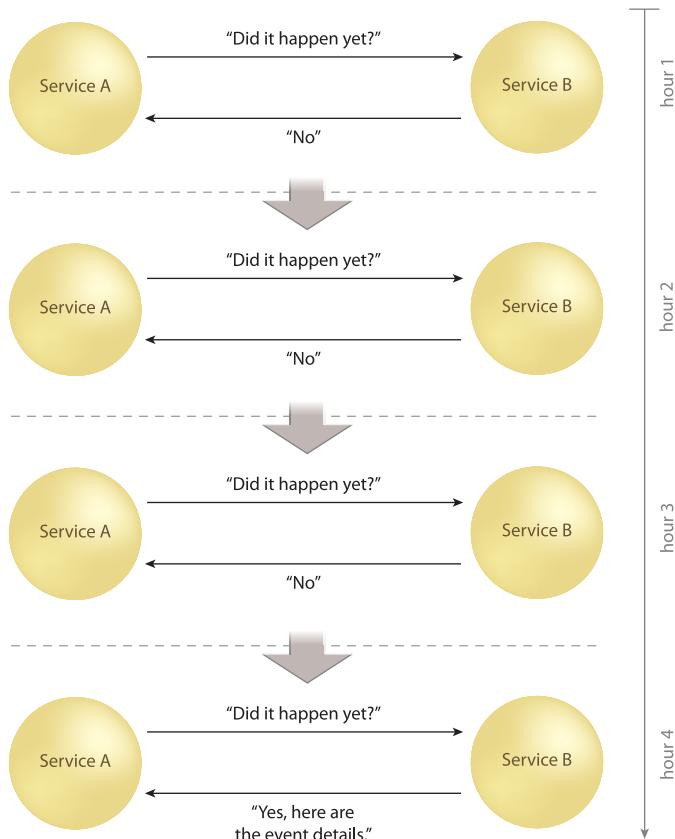


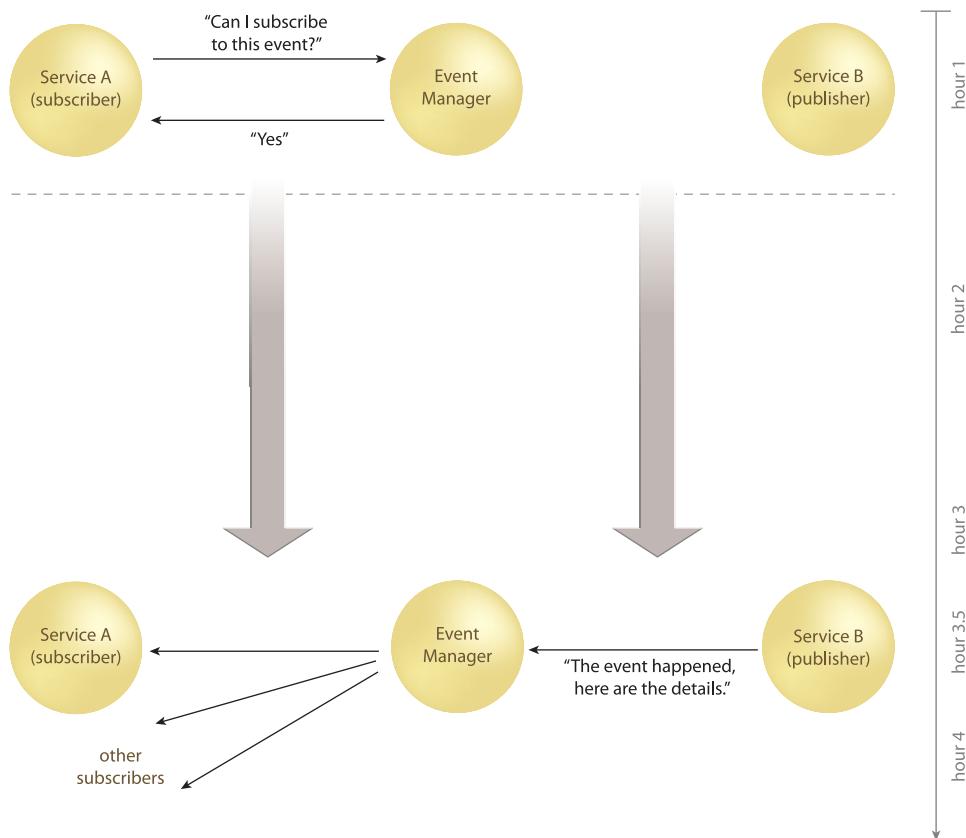
Figure 18.36

Service A (acting as a service consumer) polls Service B on an hourly basis for information about an event that Service A is interested in. Each polling cycle involves a synchronous, request-response message exchange. After the fourth hour, Service A learns that the event has occurred and receives the event information.

Solution

An event management program is introduced, allowing the service consumer to set itself up as a *subscriber* to events associated with a service that assumes the role of *publisher*. There may be different types of events that the service makes available, and consumers can choose which they would like to be subscribed to.

When such an event occurs, the service (acting as publisher) automatically sends the event details to the event management program, which then broadcasts an event notification to all of the consumers registered as subscribers of the event (Figure 18.37).

**Figure 18.37**

Service A requests that it be set up as a subscriber to the event it is interested in by interacting with an event manager. Once the event occurs, Service B forwards the details to the event manager which, in turn, notifies Service A (and all other subscribers) via a one-way, asynchronous data transfer. Note that in this case, Service A also receives the event information earlier because the event details can be transmitted as soon as they're available.

NOTE

The solution proposed by this pattern is closely related to the event-driven architecture (EDA) model. The upcoming *ESB Architecture for SOA* title that will be released as part of this book series will explore how event-driven messaging is supported via Enterprise Service Bus (704) and will further provide more detailed, ESB-specific design patterns.

Application

An event-driven messaging framework is implemented as an extension to the service inventory. Runtime platforms, messaging middleware, and ESB products commonly provide the necessary infrastructure for message processing and tracing capabilities, along with service agents that supply complex event processing, filtering, and correlation.

Impacts

Event-Driven Messaging is based on asynchronous message exchanges that can occur sporadically, depending on whenever the service-side events actually occur. It therefore may not be possible to wrap these exchanges within controlled runtime transactions.

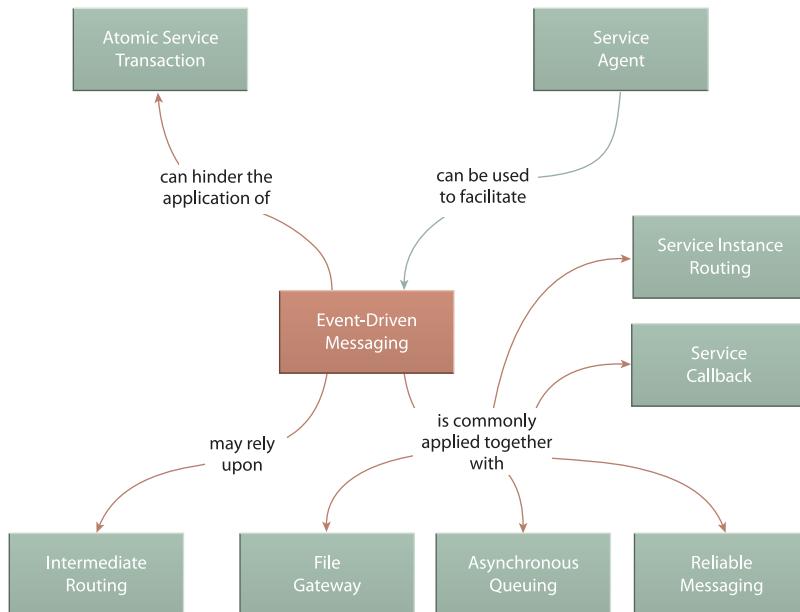
Furthermore, because notification broadcasts cannot be predicted, the consumer must always be available to receive the notification message transmissions. Also, messages are typically issued via the one-way MEP, which does not require an acknowledgement response from the consumer.

Both of these drawbacks can raise serious reliability issues that can be addressed through the application of Asynchronous Queuing (582) and Reliable Messaging (592).

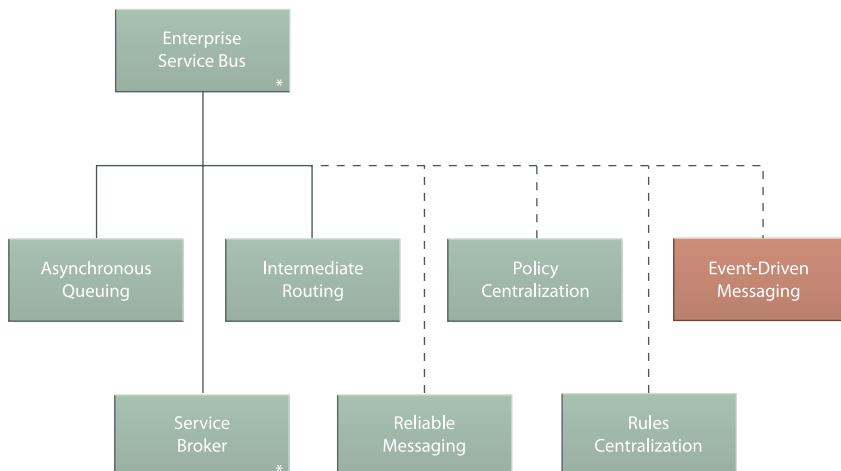
Relationships

The unique messaging model established by Event-Driven Messaging extends the base model provided by Service Messaging (533) and is itself often extended via other specialized messaging patterns, such as Asynchronous Queuing (582) and Reliable Messaging (592).

The publish-and-subscribe model that underlies Event-Driven Messaging provides advanced, asynchronous messaging functionality that can build upon the routing and messaging logic provided natively by ESB platforms (Figure 18.39).

**Figure 18.38**

Event-Driven Messaging provides distinct functionality that relies upon a combination of other messaging and agent-related patterns.

**Figure 18.39**

Event-Driven Messaging is considered an optional extension to Enterprise Service Bus (704).

NOTE

Event-Driven Messaging is broad pattern that relates to a number of established messaging patterns, including Publish-Subscribe Channel (Hohpe, Woolf) and Event Message (Hohpe, Woolf).

CASE STUDY EXAMPLE

The FRC manages a fleet of field agents that visit lumber mills and other sites owned by companies in the lumber and forestry industries. Often these visits are to follow up on complaints or concerns by company representatives. For example, the field officer may need to help educate mill operators as to how a new FRC policy affects their existing operations. However, at times field officers are also sent out to perform surprise inspections to ensure that certain procedure-related policies are, in fact, being followed.

To better support these officers, the FRC has issued mobile devices that display the agenda for a given day and any related logistical or contract information. These devices were custom-developed, and FRC architects had to make a decision as to whether they should be designed to periodically poll the central FRC scheduling system for updates or whether a push mechanism should be used instead. After interviews with relevant FRC staff (both field officers and those maintaining the internal system that contains the data required by the officers), they opted for the push approach.

Even though the periodic polling routine would have been less expensive and time-consuming to build, their interviews revealed that there are times where internal staff need to get urgent messages out to field officers. For example, in case of a natural disaster or other type of emergency. In this situation it is unacceptable to have to wait until the mobile device issues its next polling command; the notification needs to go out immediately.

To implement such a system, FRC architects turn to Event-Driven Messaging, which allows them to set up each mobile device as a subscriber to events governed by a central service acting as a publisher. Whenever relevant events occur, a message broadcast is transmitted to all field officers. The system is further able to distinguish between different types of subscribers and messages. Individual officers are subscribed to their schedule, and therefore an officer only receives an update when an agenda change affects that officer. However, other types of messages (such as those that notify about an emergency) are automatically sent out to all subscribers.

Chapter 19



Composition Implementation Patterns

Agnostic Sub-Controller

Composition Autonomy

Atomic Service Transaction

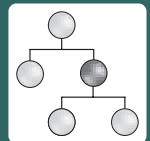
Compensating Service Transaction

Depending on the processing requirements of a service composition, certain design options can be considered as a means of improving the composition architecture. The patterns in this chapter provide a mixed set of design solutions that address implementation-level issues pertaining primarily to runtime service activity management and composition structure.

When working with task services, Agnostic Sub-Controller (607) provides a method by which sub-process logic can be reused as a nested composition structure. Composition Autonomy (616) is focused on increasing the collective performance and behavioral predictability of a service composition and is closely related to the Service Autonomy design principle. Both Atomic Service Transaction (623) and Compensating Service Transaction (631) enhance the integrity of runtime service activities by wrapping them in coordinated boundaries.

Agnostic Sub-Controller

How can agnostic, cross-entity composition logic be separated, reused, and governed independently?



Problem	Service compositions are generally configured specific to a parent task, inhibiting reuse potential that may exist within a subset of the composition logic.
Solution	Reusable, cross-entity composition logic is abstracted or made accessible via an agnostic sub-controller capability, allowing that subset of the parent composition logic to be recomposed independently.
Application	A new agnostic service is created or a task service is appended with an agnostic sub-controller capability.
Impacts	The addition of a cross-entity, agnostic service can increase the size and complexity of compositions and the abstraction of agnostic cross-entity logic can violate modeling and design standards established by Service Layers (143).
Principles	Service Reusability, Service Composability
Architecture	Composition, Service

Table 19.1

Profile summary for the Agnostic Sub-Controller pattern.

NOTE

The term “sub-controller” is not new to this pattern. It is used to represent any service or service capability that composes services and is itself also composed. See SOAGlossary.com for a full definition.

Problem

When following Non-Agnostic Context (319), non-agnostic logic, at the time it is originally defined, is considered single purpose and non-reusable. By applying Process Abstraction (182), this type of logic is isolated within a task service that is typically positioned as the parent controller of a composition. When also having services based upon Entity Abstraction (175), the task service will generally be comprised of composition logic that needs to span multiple business entity boundaries in order to compose the respective entity services.

Subsequent to its definition and usage, it may be discovered that the task service contains segments of cross-entity logic that are, in fact, agnostic. This type of logic is most comparable to sub-processes that represent composition sequences so common that they are considered reusable. However, because this logic resides embedded within a larger body of non-agnostic logic, its reuse potential cannot be realized (Figure 19.1).

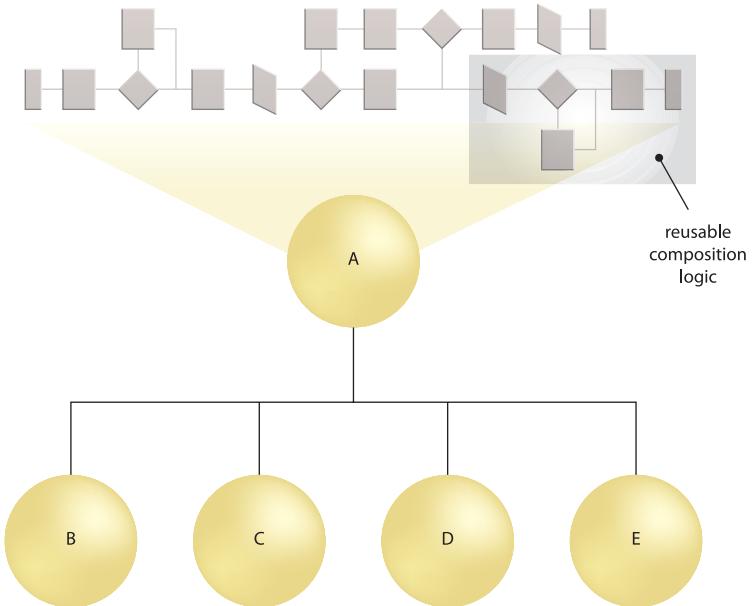
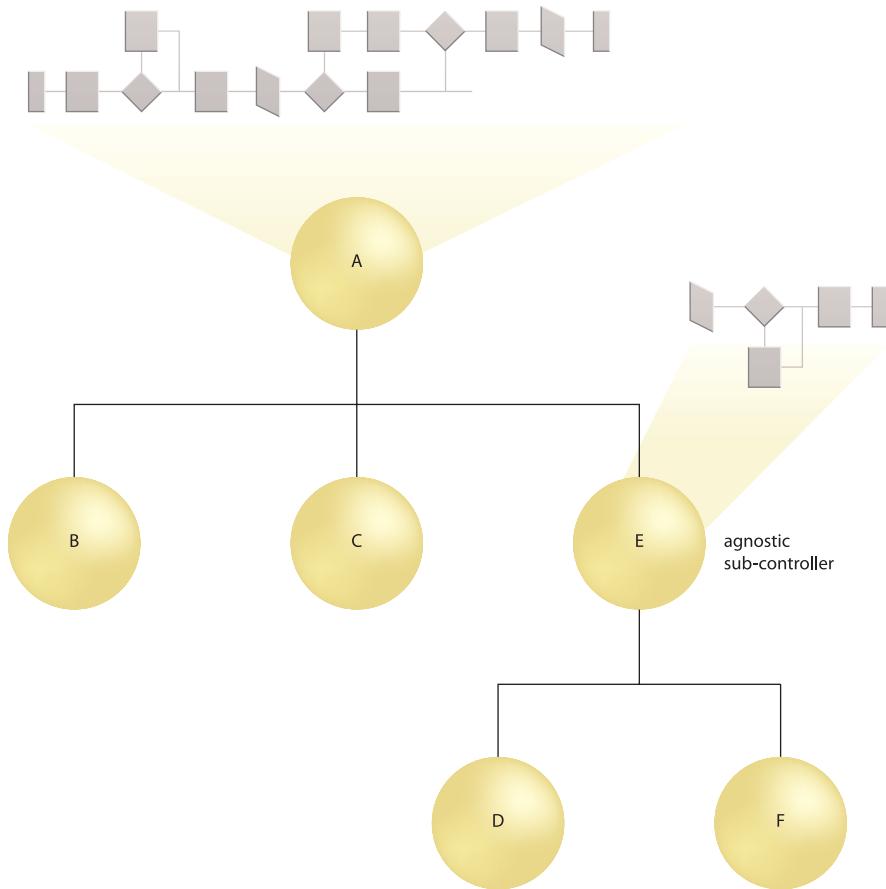


Figure 19.1

A subset of the parent process logic is deemed reusable but it is trapped among the other non-agnostic process logic encapsulated by the task service.

Solution

The newly discovered, cross-entity agnostic logic is separated into an agnostic service or an agnostic capability is added to the original task service. Either way, the result is a body of composition logic that itself can now be independently recomposed, thereby establishing an agnostic sub-controller (Figure 19.2).

**Figure 19.2**

A traditional, single-purpose composition is often configured in a two-tier hierarchy, with all of the composition logic residing in the parent task service. Alternatively, the composition can be structured into additional tiers so that the composition is comprised of a parent controller service and one or more nested compositions represented by sub-controller services. These nested compositions may be necessary to carry out the parent task, but individually they can also provide logic that can be used independently to automate a smaller task, or they may have logic that can be used to automate other larger tasks. Either way, they can be structured to represent and abstract agnostic logic for reuse purposes.

Application

There are two common methods for abstracting agnostic, cross-entity logic, each with its own set of trade-offs:

- *New Agnostic Service* – The logic forms the basis of a new agnostic service. The challenge may lie in positioning this service within any of the layers previously established by Service Layers (143). Because it will usually represent cross-entity logic, it may be possible to assign it a coarse-grained business entity-based scope, thereby qualifying it as a standalone entity service.
- *New Agnostic Capability* – The logic remains within the task service but it is made accessible via a new capability exposed by the task service contract. Although a practical approach, this can disturb the clean separation previously achieved by Process Abstraction (182).

With either technique, supporting infrastructure will need to be upgraded to prepare for the reuse of the newly found agnostic logic.

Impacts

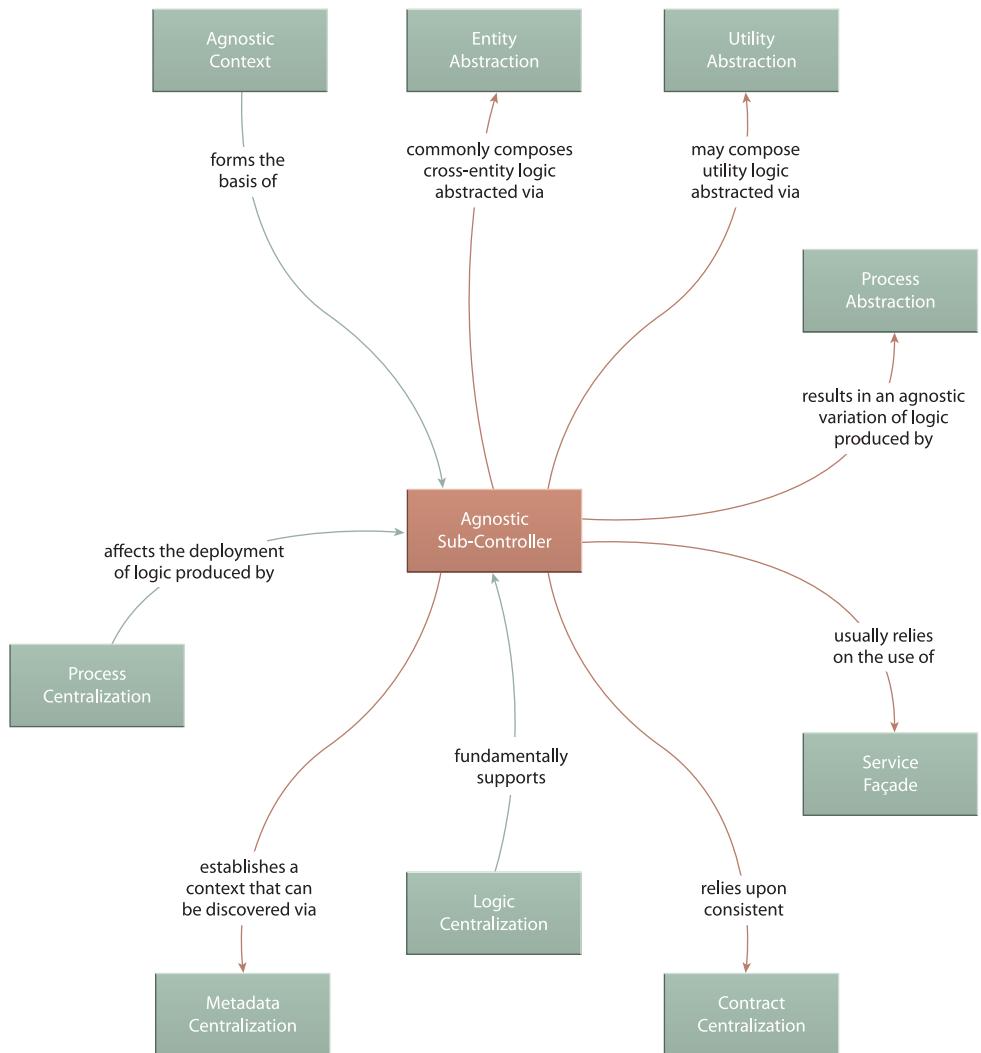
The main challenge with the application of this pattern is its effect on the overall logical structure of the service inventory. Depending on the scope and nature of the agnostic sub-controller logic, its isolation into a separate service or capability can easily violate one or more of the patterns associated with Service Layers (143). The resulting confusion and misalignment of functional service contexts may not be warranted.

It is recommended that the actual reuse potential of the identified logic be first established and confirmed before proceeding with this pattern.

Relationships

Agnostic Sub-Controller is applied to service logic that is likely to have previously been non-agnostic, which is why this pattern has relationships with Process Abstraction (182), Process Centralization (193), and Agnostic Context (312).

Service Façade (333) is often employed to help structure the internal logic of an agnostic sub-controller service, and Contract Centralization (409) is important and relevant to the successful long-term usage of this type of service (as with any service).

**Figure 19.3**

Agnostic Sub-Controller is in the unique position of having relationships with patterns focused on non-agnostic and agnostic logic.

CASE STUDY EXAMPLE

The original design of the Alleywood Report Zones task service encapsulates all the composition logic required to assemble a consolidated report by retrieving data from the Region, Area, and Policy Checks entity services, as shown in Figure 19.4.

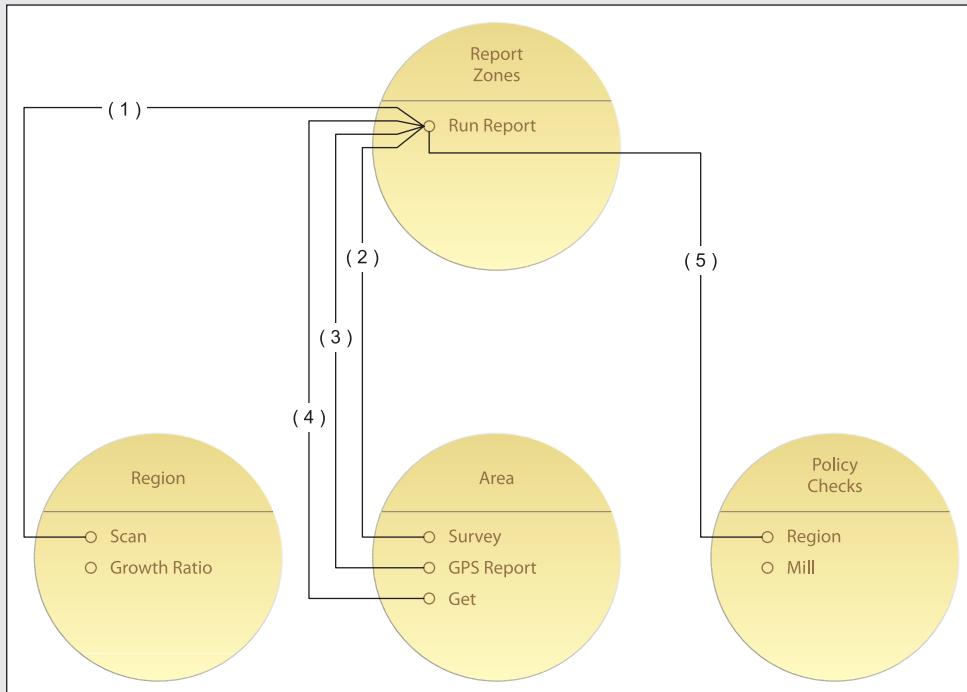


Figure 19.4

The initial service composition hierarchy established by the Report Zones task service responsible for carrying out the Report Zones business task.

This composition automates the Report Zones business process, which collects a range of information about the regions and areas that fall within a particular zone and then cross-references this data with any relevant FRC issued policies.

The process is comprised of the following steps:

1. Perform a scan of one or more regions to identify all affected areas.
2. For each area, carry out a zoning survey and retrieve the most current GPS report data. Based on the results of these queries, request the corresponding area records.

3. Consolidate the area record data by establishing a special, temporary view record comprised of textual and image information.
4. Perform policy checks for all area records to ensure that any FRC policies that may affect an area are identified.

As illustrated in Figure 19.4, the Report Zones service invokes the Scan capability of the Region service to collect region data and then invokes three separate capabilities from the Area service to further gather Area information that pertains to the previously identified regions. It then carries out its own embedded logic to establish a custom view of the area data before passing information over to the Policy Checks utility service.

During a meeting with a Tri-Fold project team, Alleywood analysts hear about a Geo-Analysis solution they are currently designing that requires similar reporting functionality. Only, the Tri-Fold task service already receives region data as input values passed to the task service's capability. The Tri-Fold team is therefore planning to build composition logic that queries the Area and Policy Checks services only, as shown in Figure 19.5.

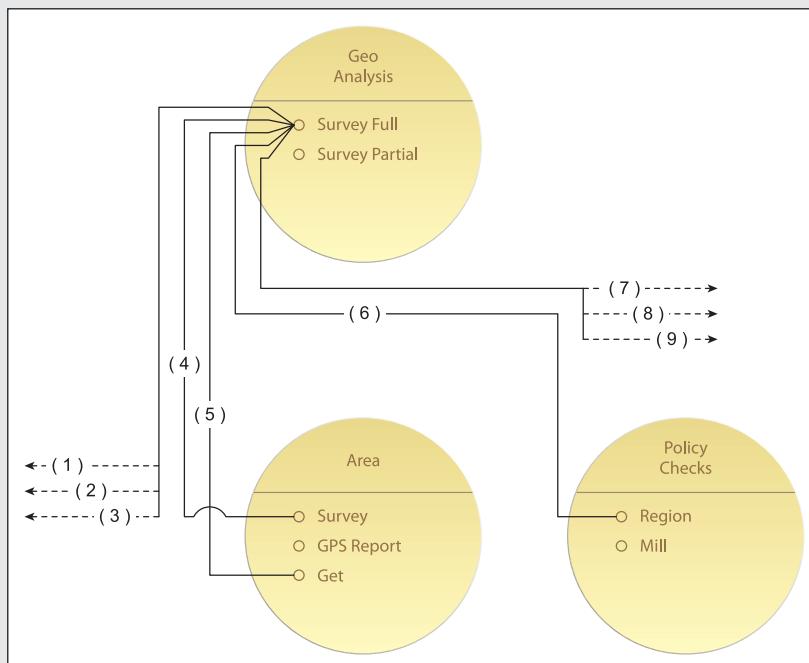


Figure 19.5

The Area and Policy Checks services are invoked as part of a larger composition involving six other services (not shown).

Another difference between the Report Zones and Geo-Analysis tasks is that the latter does require image data and therefore does not invoke the Area service's GPS Report capability.

However, after some discussion among the project teams, it is proposed that there is sufficient common functionality to warrant the creation of an agnostic sub-controller, especially given that additional reuse opportunities for this logic are also anticipated. This new agnostic Area Policy Report service is created. It abstracts the collection and consolidation of area data and the retrieval of corresponding policies, as shown in Figure 19.6.

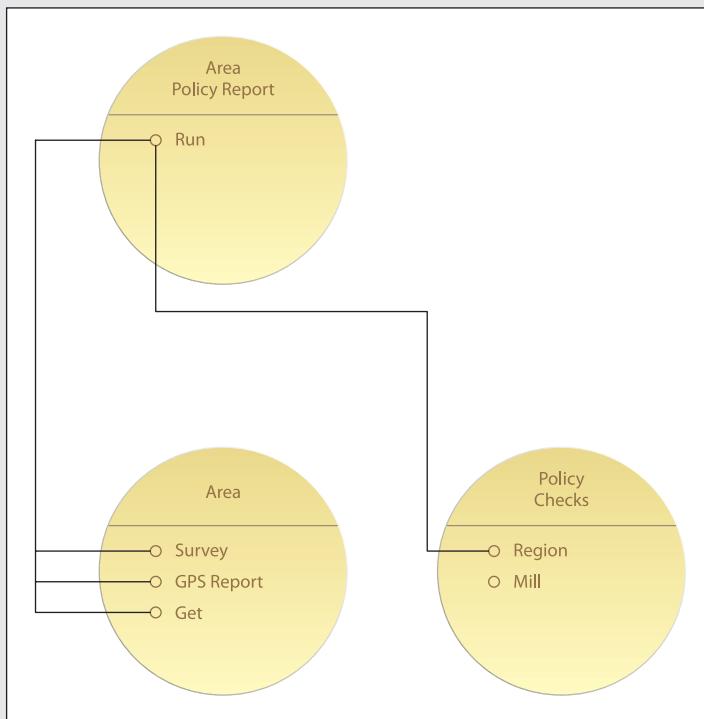


Figure 19.6

The new Area Policy Report service isolating the recently discovered agnostic, cross-entity logic. The relationship lines between capabilities are not numbers because the capabilities the service invokes and the sequence in which they are invoked is determined by the input value received via its Run capability.

At first there is some resistance to the idea of introducing a new service into the inventory only to provide this one small body of agnostic composition logic. A debate ensues as to whether it would make more sense to simply add a Policy Report capability to the Area service or to perhaps create a new Policy entity service with an Area Report capability. By reviewing the service inventory blueprint, the teams discover that a Policy entity service has in fact been modeled and is planned for delivery later in the year.

However, subsequent discussions lead to the conclusion that a separate agnostic service is the best design option for this case. It allows the reporting functionality to be abstracted and extended across two different entity services, as required. Besides being able to locate this logic in a separate physical service implementation, concerns regarding how this cross-entity logic could be incorporated within existing entity services (without violating established entity service boundaries) is avoided by simply classifying it as a utility service.

As it enters the design phase, the Area Policy Report service is subjected to the same design rigor as any other agnostic service. The result is a flexible set of composition logic encapsulated by the Run capability, allowing the service to receive a range of input values. This, for example, enables this agnostic sub-controller to support both the Report Zones and Geo-Analysis tasks by providing optional GPS image data to the former (Figure 19.7).

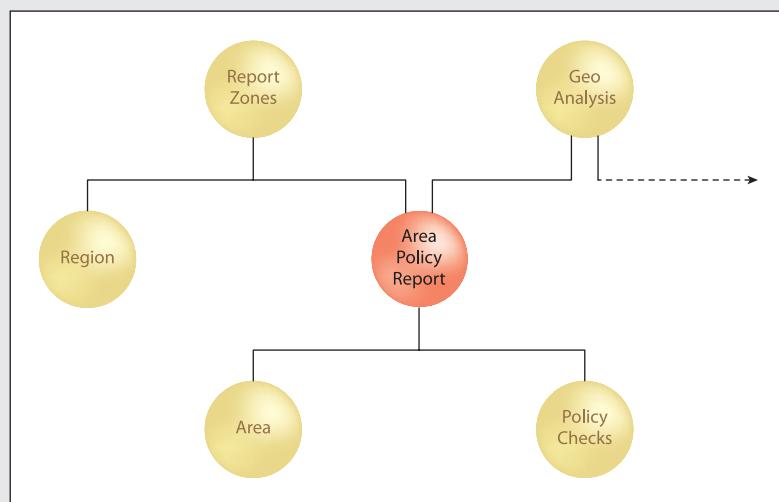
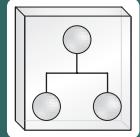


Figure 19.7

The agnostic Area Policy Report service being shared by the Report Zones and Geo-Analysis compositions.

Composition Autonomy

How can compositions be implemented to minimize loss of autonomy?



Problem	Composition controller services naturally lose autonomy when delegating processing tasks to composed services, some of which may be shared across multiple compositions.
Solution	All composition participants can be isolated to maximize the autonomy of the composition as a whole.
Application	The agnostic member services of a composition are redundantly implemented in an isolated environment together with the task service.
Impacts	Increasing autonomy on a composition level results in increased infrastructure costs and government responsibilities.
Principles	Service Autonomy, Service Reusability, Service Composability
Architecture	Composition

Table 19.2

Profile summary for the Composition Autonomy pattern.

Problem

Services are ideally individually autonomous so that they can provide a high degree of behavioral predictability when repeatedly reused and shared across multiple compositions. However, a natural result of typical distributed service composition is a *loss of autonomy* by any service composing another simply due to the fact that the service is required to invoke solution logic that resides outside of its controlled execution environment.

When individual services provide high levels of individual autonomy, the collective autonomy of the composition is correspondingly elevated, and this is generally adequate for most service compositions. But when one or more composition participants have a poor level of autonomy or when the requirements of the composition as a whole demand a higher degree of robustness and reliability, then a distributed composition may be insufficient (Figure 19.8).

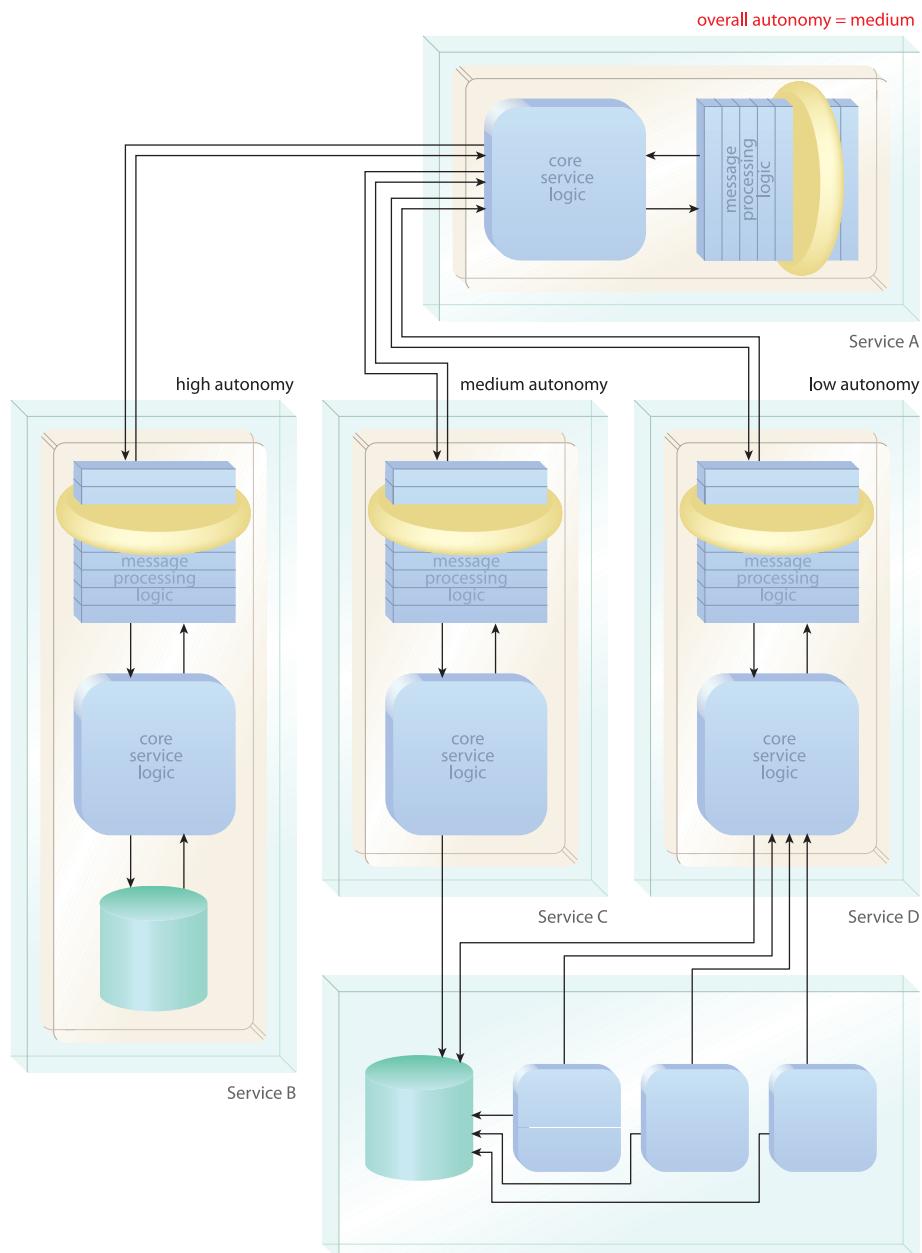


Figure 19.8

In a highly distributed environment, a service composition's overall autonomy will be collectively determined by the autonomy of its individual composition members. When services are designed in accordance with the Service Autonomy design principle, this collective autonomy will generally be sufficient. However, there are times when a higher degree of composition autonomy is required. As shown in this figure, Service D offers the lowest level of autonomy because its logic is accessed by external programs.

Solution

The services participating in the composition are deployed within an isolated environment so as to give the composition as a whole a high level of autonomy (Figure 19.9).

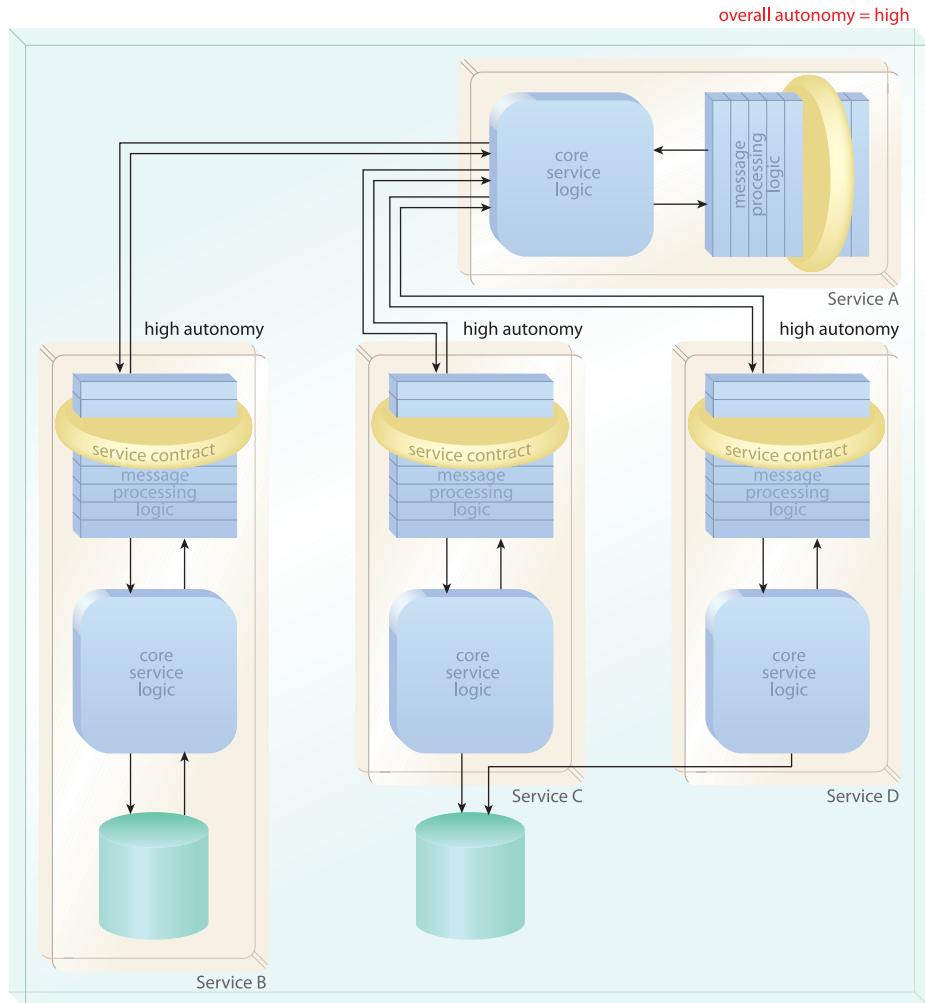


Figure 19.9

By grouping the services of a composition into a separate deployment environment, the collective autonomy is maximized because the implementation is dedicated to the composition, and none of the services are otherwise shared. Services C and D in particular benefit from this new implementation as they are no longer subject to shared access.

Application

To increase the autonomy of a composition the following steps can be taken:

- Two or more composed services are deployed on the same physical server to avoid remote communication.
- The composed services are isolated to avoid shared access.
- The agnostic composed services are redundantly implemented, as per Redundant Implementation (345).
- Services requiring data access are supplied with dedicated or replicated data stores, as per Service Data Replication (350).

It's important to keep in mind that a service composition is a design-time representation of the aggregated service capabilities required to automate a specific business task. This has the following implications when applying this pattern:

- Even though entire services are redundantly deployed, only a subset of the capabilities will likely be utilized by the composition.
- Depending on the complexity of the business task, a variety of runtime scenarios can result in different capability compositions, some of which may need to involve services outside of the dedicated composition environment.

NOTE

There are variations in the extent to which this design pattern can be applied, several of which correspond to autonomy levels described in Chapter 10 of *SOA Principles of Service Design*.

Impacts

Because the requirements for increasing the overall autonomy of a service composition usually involve upgrading or extending the existing enterprise infrastructure, there are obvious costs and impacts that need to be taken into consideration.

Redundant implementations of agnostic services further add governance effort and expense so that these independent implementations are kept in sync with their shared counterparts.

Relationships

Because of its narrow emphasis on supporting the goals of the Service Autonomy design principle, this pattern has just a handful of relationships. Fundamentally, Composition Autonomy is concerned with maximizing the reliability and availability of services based on Agnostic Context (312).

Service Data Replication (350) and Redundant Implementation (345) help establish an appropriate level of isolation for service compositions. Often, security needs will dictate the decision to isolate externally facing services and any services they may compose, which may lead to the need to apply Composition Autonomy in support of Inventory Endpoint (260).

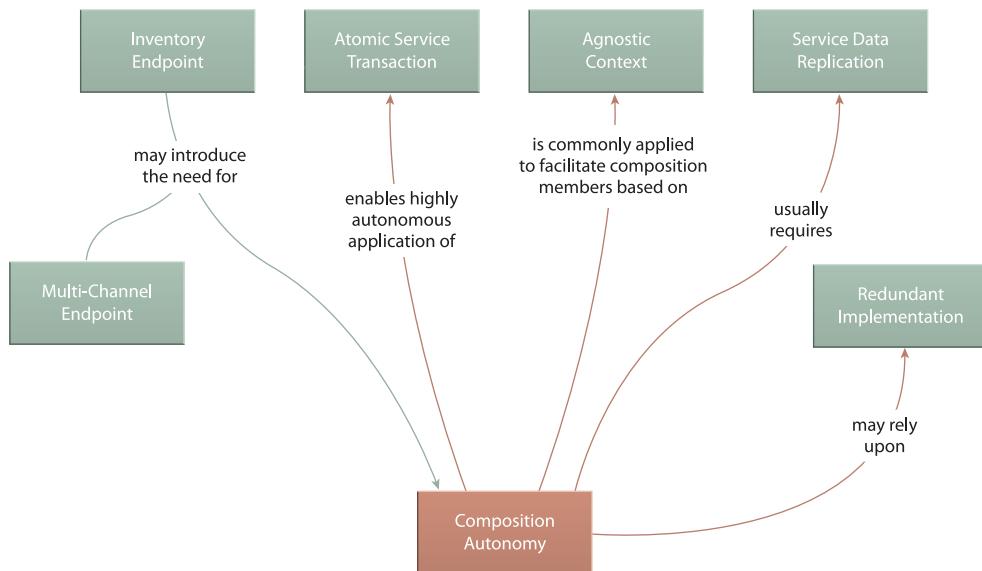


Figure 19.10

Composition Autonomy is implementation-centric and will generally require the application of more specialized patterns to realize required levels of composition isolation.

CASE STUDY EXAMPLE

Since it was first implemented, the Area Policy Report composition (described earlier in the Agnostic Sub-Controller (607) example) has become increasingly popular. Six separate parent compositions now utilize its relatively simple functions to receive consolidated area and policy reports. Each of these parent compositions may, at any time, invoke multiple instances of this service.

Initially there were no problems as the requested reports were relatively small and the usage demands were manageable. However, the request for larger, more complex reports has increased, as has the extent of concurrent usage. On any given day, a peak usage of over 60 instances of the Area Policy Report service is reached.

These demands have predictably resulted in a noticeable increase in latency, which has led to a need to better optimize this service. Upon investigating the physical composition architecture, it is discovered that the Area Policy Report task service is located on a different server than the Area and Policy Checks services it is responsible for composing (Figure 19.11).

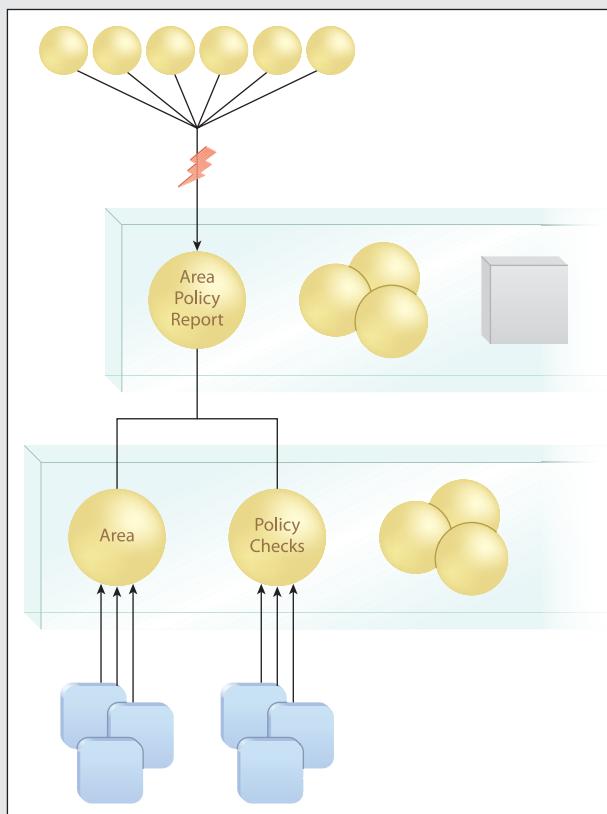


Figure 19.11

All services reside in shared processing environments, thereby reducing the overall autonomy of the composition.

The Area Policy Report service is co-located with eight other task services and a small legacy application, all on one physical server. Being more established agnostic resources, the Area and Policy Checks services have been deployed on a powerful, multi-processor server. Yet, as agnostic resources, these services are individually reused by many other compositions.

Upon further review of usage statistics, analysts identify the Area entity service as the primary bottleneck. Sometimes the Area Policy Report service is required to call the Area service more than once for a single report, plus the quantity of data that needs to be retrieved by the Area service can sometimes be significant (especially when large amounts of GPS image data are requested).

Because there are several solution delivery projects underway that have identified the need to compose the Area Policy Report service, the demand for this composition will only increase. As a result, the decision is made to apply Composition Autonomy, but only to a limited extent.

A dedicated server is set up, allowing a redundant implementation of the Area service to be deployed together with the sole implementation of the Area Policy Report service. The Policy Checks utility service is not part of this new isolated environment at this point, as it continues to perform within acceptable parameters. However, the option is always there for a redundant implementation of this service to also be added to this environment (Figure 19.12).

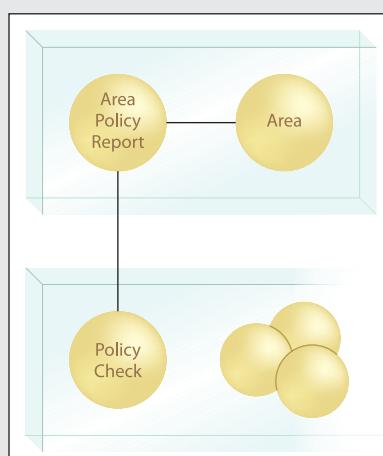
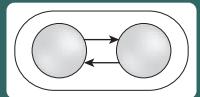


Figure 19.12

The new environment dramatically increases composition autonomy by isolating the two services responsible for performing the bulk of the composition logic.

Atomic Service Transaction

How can a transaction with rollback capability be propagated across messaging-based services?



Problem	When runtime activities that span multiple services fail, the parent business task is incomplete and actions performed and changes made up to that point may compromise the integrity of the underlying solution and architecture.
Solution	Runtime service activities can be wrapped in a transaction with rollback feature that resets all actions and changes if the parent business task cannot be successfully completed.
Application	A transaction management system is made part of the inventory architecture and then used by those service compositions that require rollback features.
Impacts	Transacted service activities can consume more memory because of the requirement for each service to preserve its original state until it is notified to rollback or commit its changes.
Principles	Service Statelessness
Architecture	Inventory, Composition

Table 19.3

Profile summary for the Atomic Service Transaction pattern.

Problem

During the course of a runtime service activity, a variety of issues can arise, relating either to the delivery of data between participating services or to problems occurring within service boundaries.

If a serious failure condition is encountered and insufficient exception handling logic is available within all affected services, then the overarching business process will not be allowed to complete successfully, nor will it be allowed to complete a pre-defined failure condition.

Instead, services may simply be left hanging in suspension indefinitely or until they time out. Outstanding changes that some services may have made to databases or other resources can end up causing problems or even corrupting parts of the enterprise because other required changes were never completed (Figure 19.13).

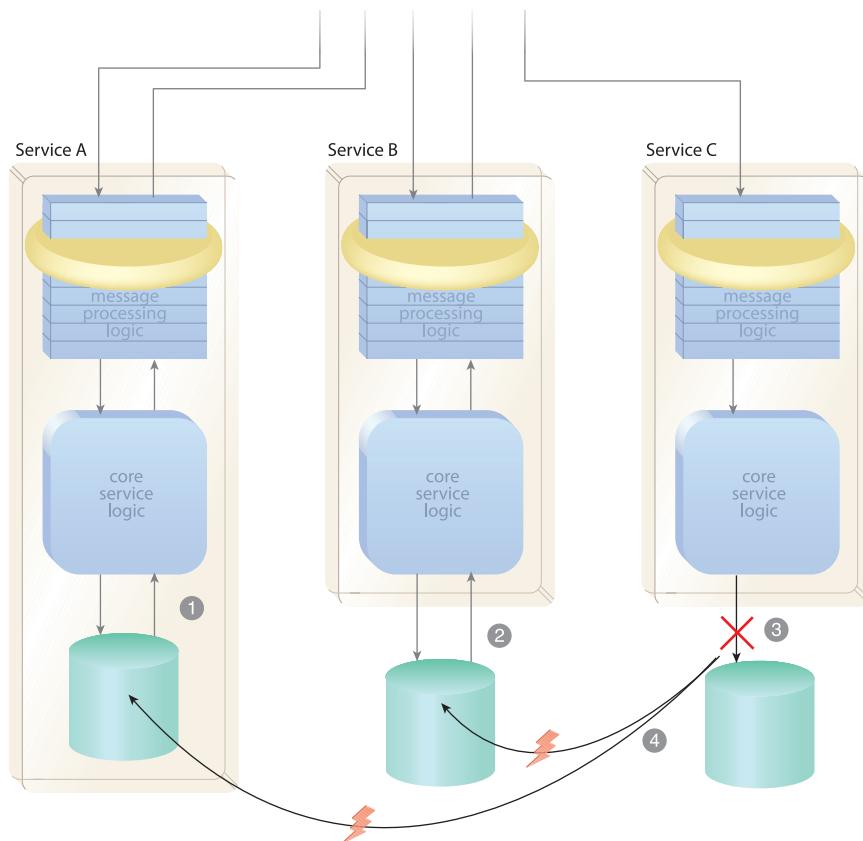


Figure 19.13

This figure shows three services being sequentially invoked in support of automating a parent business task. The first two services successfully complete their required database updates (1, 2), but the third service fails at its attempt to update its database (3). According to the rules of the business process, either all three updates must succeed or none at all. However, because the first two updates have already been completed, the failed update of the third database compromises the quality of the data in the first two (4).

Solution

A transaction system can require that services within a particular composition register themselves as part of a transaction prior to completing their changes. Then, as the service activity is underway, participating services communicate with the transaction system as to their status (Figure 19.14).

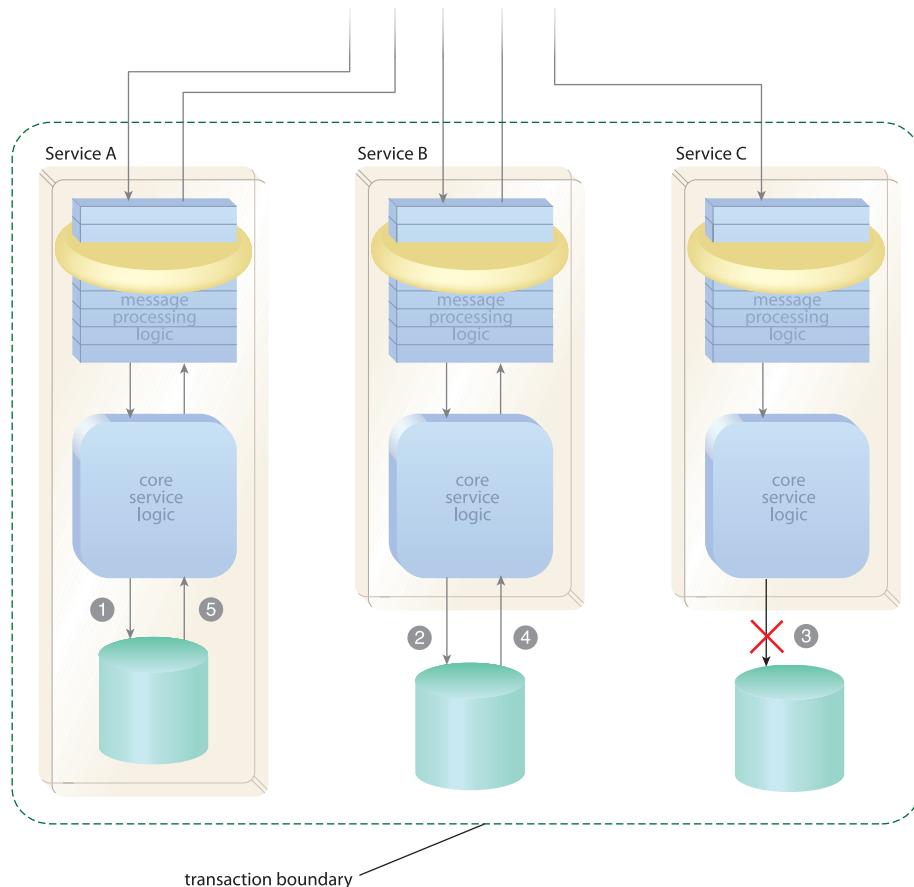


Figure 19.14

As per the previous figure, Services A and B complete their respective tasks successfully. However each time they do, they initiate a local transaction, temporarily saving the current state of the database prior to making their changes (1, 2). After Service C fails its database update attempt (3), Services A and B restore their databases back to their original states (4, 5). The business task is effectively reset or rolled back across services within the pre-defined transaction boundary.

At some point, the system queries all services to ask whether their functions were carried out successfully. If any one service responds negatively (or if any one service does not respond at all), a rollback command is issued, requesting that all services now restore any changes made up until that point. However, if all services respond favorably, then a commit command is issued asking all services to commit their changes.

Application

Depending on the transaction management system being used, the application of this design pattern can vary. Fundamentally, though, some mechanism needs to be in place so that all services within a given composition can be tracked at runtime and then contacted to receive status updates and for notification of commit or rollback commands.

For services delivered as components, runtime transaction management systems are capable of natively establishing transaction boundaries across services, especially when they rely on traditional binary protocols that create persistent connections.

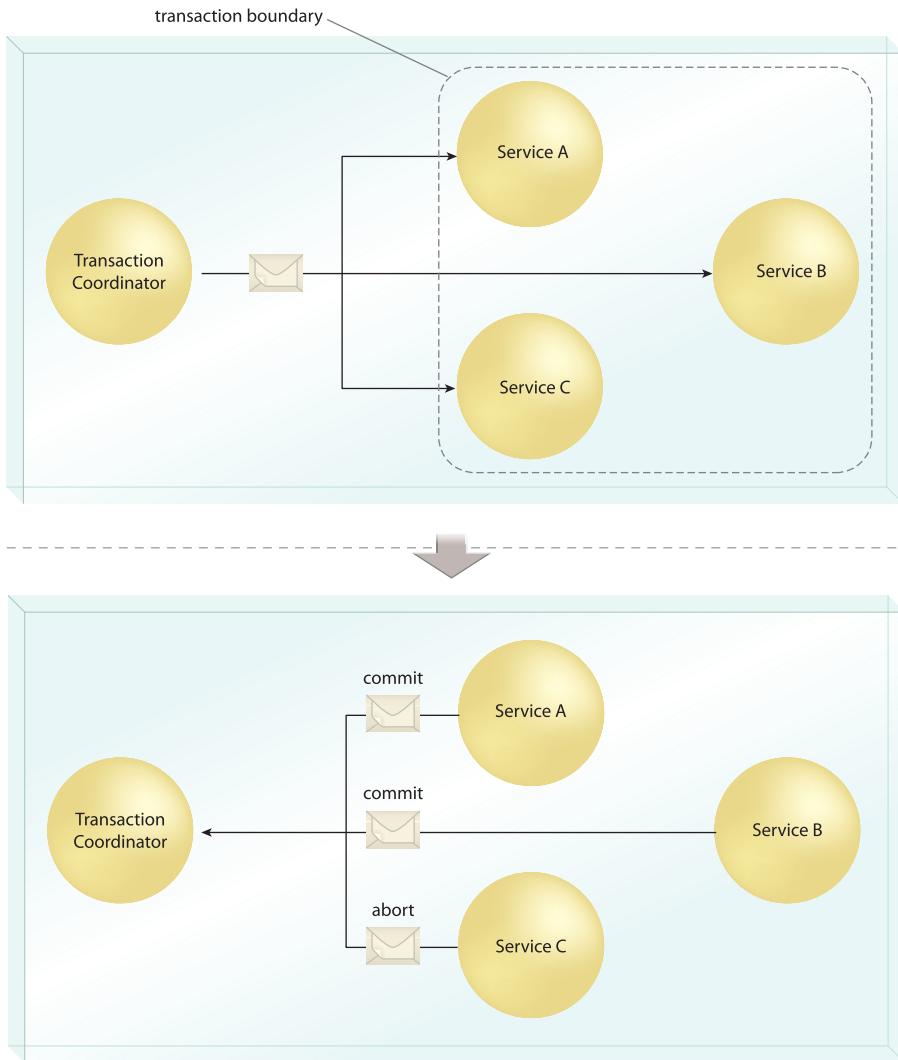
When services are built as Web services, the WS-Coordination and WS-AtomicTransaction standards provide an industry standard mechanism to support transaction propagation across Web service implementations. Figure 19.15 illustrates how, when using WS-Coordination, a coordinator service is positioned to define the transaction boundary and manage transaction activity.

NOTE

For more information about WS-Coordination and related standards, visit SOASpecs.com and WS-Standards.com and see Chapter 6 in *Service-Oriented Architecture: Concepts, Technology, and Design*.

Impacts

For services to effectively participate in an atomic transaction, they need to capture a snapshot of a resource prior to making changes to it. This previous condition is often loaded into memory as state data and will continue to consume memory until the service receives the commit or rollback command. In larger transactions involving multiple services, this amount of memory consumption can add up and reduce overall service scalability (an issue especially important to shared agnostic services).

**Figure 19.15**

The transaction coordinator establishes the transaction boundary as per participating services that register for this transacted activity. These services are then queried as to whether the transaction should be committed or rolled back. Because Service C's database update attempt failed, it votes to abort the transaction, which is what subsequently happens.

Relationships

This pattern can touch on many runtime activity-related parts of an inventory architecture, which is why it has so many relationships with patterns concerned with messaging, agent, and composition processing.

Atomic transactions are often initiated and coordinated via the parent business process layer as per Process Abstraction (182) and are extra effective when all participants can be physically isolated as per Composition Autonomy (616).

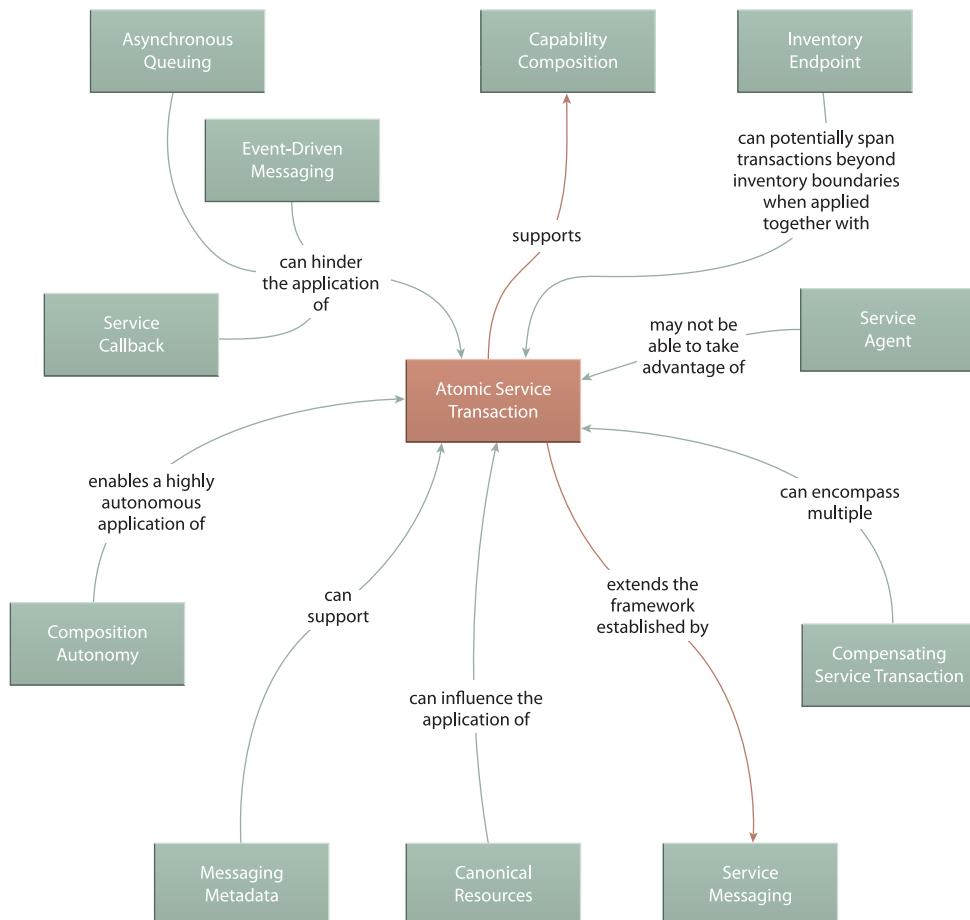


Figure 19.16

Atomic Service Transaction establishes a framework for coordinating transactions that involve numerous services and therefore relates to a variety of other patterns that provide runtime processing features.

However, asynchronous-based messaging patterns, such as Event-Driven Messaging (599) and Asynchronous Queuing (582), are often incompatible with this pattern due to its need to lock resources and its common reliance on synchronous data exchanges.

When centralizing process logic as part of an orchestration environment, the scope and complexity of service compositions can increase, primarily due to the rich feature set usually provided by orchestration products. Orchestrated logic therefore benefits from the runtime control provided by Atomic Service Transaction, which is why this pattern represents a common extension to Orchestration (701).

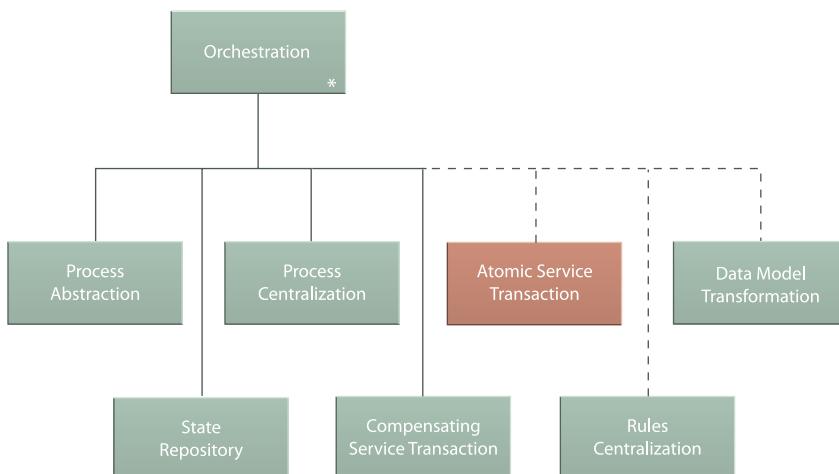


Figure 19.17

Atomic Service Transaction represents an optional part of Orchestration (701).

CASE STUDY EXAMPLE

Several service compositions created in support of the Alleywood and Tri-Fold accounting systems require that either all parts of the composition succeed or fail. Through the use of transaction metadata, messages can be equipped with header blocks that express transaction details, as shown in Example 19.1. This functionality effectively allows a transaction to span numerous Web services.

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"  
    xmlns:wscoor="http://schemas.xmlsoap.org/ws/2002/08/wscoor"  
    xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">  
    <Header>  
        <wscoor:CoordinationContext>  
            <wsu:Expires>
```

```
2009-03-03T00:00:00.0000000-09:00
</wsu:Expires>
<wsu:Identifier>
  uuid:isidf843249-454580-dfs
</wsu:Identifier>
<wscoor:CoordinationType>
  http://schemas.xmlsoap.org/ws/2003/09/wsat
</wscoor:CoordinationType>
</wscoor:CoordinationContext>
</Header>
<Body>
  ...
</Body>
</Envelope>
```

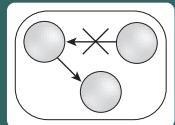
Example 19.1

This header block establishes that the `Expires` and `Identifier` element values are associated with an atomic transaction (as defined in the `CoordinationType` element).

These are just some of the examples of technologies that currently support industry standard messaging metadata. Furthermore, these metadata types can be combined into the same header section, allowing each message to be outfitted with a rich set of meta information types.

Compensating Service Transaction

By Clemens Utschig-Utschig, Berthold Maier, Bernd Trops, Hajo Normann, Torsten Winterberg, Brian Loesgen, Mark Little



How can composition runtime exceptions be consistently accommodated without requiring services to lock resources?

Problem	Whereas uncontrolled runtime exceptions can jeopardize a service composition, wrapping the composition in an atomic transaction can tie up too many resources, thereby negatively affecting performance and scalability.
Solution	Compensating routines are introduced, allowing runtime exceptions to be resolved with the opportunity for reduced resource locking and memory consumption.
Application	Compensation logic is pre-defined and implemented as part of the parent composition controller logic or via individual “undo” service capabilities.
Impacts	Unlike atomic transactions that are governed by specific rules, the use of compensation logic is open-ended and can vary in its actual effectiveness.
Principles	Service Loose Coupling
Architecture	Inventory, Composition

Table 19.4

Profile summary for the Compensating Service Transaction pattern.

Problem

When services being composed at runtime encounter failure conditions, runtime exceptions are raised either by individual services or by the platform hosting the service composition instance.

Services may or may not have adequate internal logic to gracefully handle some exceptions, thereby preserving the integrity of the remaining composition. However, with serious or unanticipated exceptions, services often have no choice but to propagate the exception to other services in the composition. Similarly, when the runtime platform itself generates an exception, it may be required to halt the execution of the composition or terminate the

composition instance altogether. One approach to solving this problem while preserving the integrity of the composition is the usage of transactions.

When applying Atomic Service Transaction (623), these situations are resolved by enabling all services to rollback to their original state before the composition instance is destroyed. In order to obtain this type of functionality, back-end resources (such as database tables and records, as well as other systems) will typically need to be locked for the duration of the transaction (Figure 19.18).

Depending on the length and scope of the transaction, this can severely degrade performance and can also reduce the scalability of the overall service inventory.

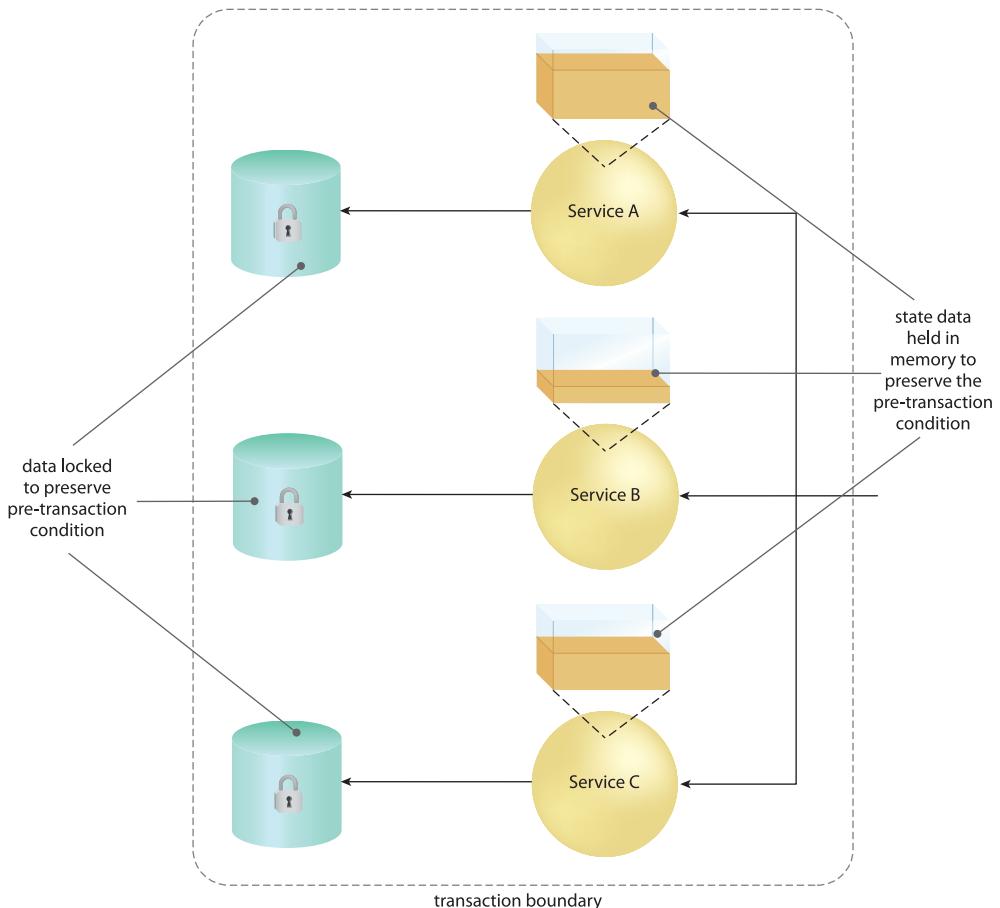


Figure 19.18

Services involved in an atomic transaction are required to lock up resources for the duration of the transaction.

Solution

Instead of being wrapped in an atomic transaction, the service composition is supplemented with compensating logic. This logic differs from atomic transactions in that it does not require services to maintain the original state or lock resources for the duration of the transaction.

As long as there is no firm requirement for compensations to restore a runtime activity back to its original state, exception conditions can be handled gracefully without jeopardizing the integrity of the composition (Figure 19.19).

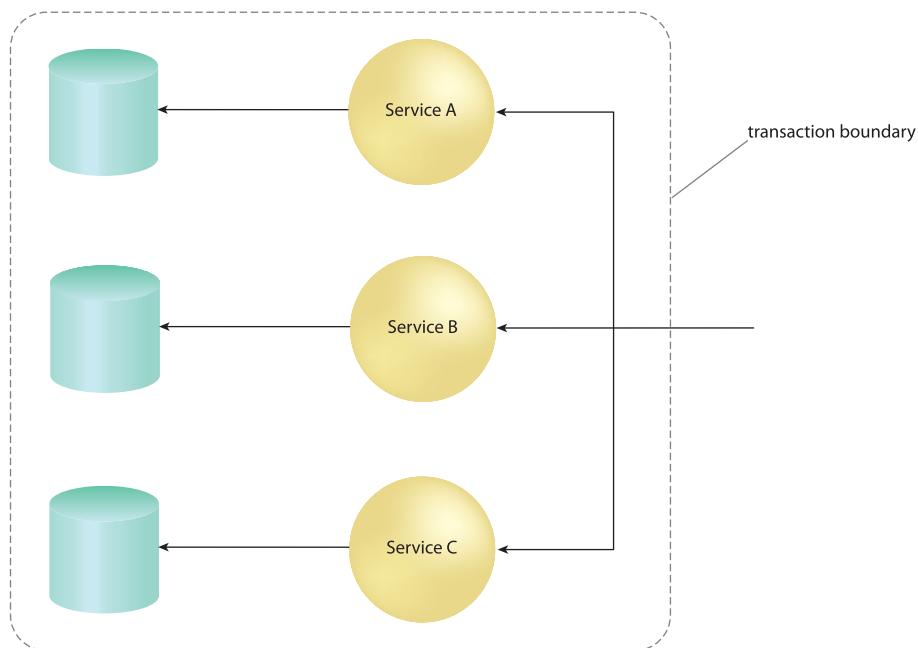


Figure 19.19

Compensating transactions do not require that resources be locked or that the original state be preserved.

Application

Compensating logic is typically defined at these levels:

- composition member
- composition controller

A service is considered a composition member when it is being composed by parent composition logic. At this level, compensating routines can extend a service via the addition of *undo* capabilities. For every capability that alters the state of the underlying service implementation (including any repositories or other resources accessed by the service logic), a corresponding undo capability is added. This allows the parent composition logic to respond to an exception condition by calling the undo capability of each service that was invoked up until that point.

The composition controller may be a task service that is composing other services or it may represent parent composition logic that is part of a platform based on Orchestration (701). When applied within middleware environments, the role of a composition controller is comparable to that of a transaction manager.

Either way, this logic can be extended with compensating routines that also respond to exception conditions with pre-determined exception handling logic. The only difference is that, in this case, the logic resides with the task logic, as per Process Abstraction (182).

At the controller level, the compensating logic may optionally invoke the individual undo capabilities of composition member services. It can also be designed to carry out additional functions, such as sending out a notification message or starting up a new instance of the composition after the previous instance was terminated. In most cases, compensation logic will invalidate the last “committed” state, as opposed to canceling back to the original pre-state.

The undo logic can vary from carrying out a simple notification to restoring some or all of the resources modified by the service. Another approach is to have the undo capability simply tag affected resources (such as an updated database record) with a marker that indicates its invalidation.

When redundant service capabilities are appended to a contract via Contract Denormalization (414), correspondingly redundant undo capabilities (that provide undo functionality at different levels of granularity) can also be added.

NOTE

When designing services as Web services and applying Orchestration (701), the WS-BPEL language provides a common means of defining compensating processes as part of the overall parent composition logic. For more information about WS-BPEL, visit SOASpecs.com and see Chapters 6 and 16 in *Service-Oriented Architecture: Concepts, Technology, and Design*.

Impacts

Because the nature and extent of compensating logic is left up to the designer, it can vary in quality and effectiveness. Creating a series of undo capabilities that perform little or no logic that actually undoes anything can lead to a false assumption that an exception was effectively handled, when in fact it wasn't. This is especially the case when composition designers are not allowed access to the underlying logic of composition member services and must therefore simply assume that undo capabilities provided by service contracts are handling failure conditions properly.

Additionally, for services that do perform a range of data or resource access functions, having to append the service contract with an undo capability for each existing capability that alters the service implementation state can bloat the service contract.

Relationships

Compensating Service Transaction shares several of the same relationships as Atomic Service Transaction (623) and also has its own relationship with that pattern as any one compensating transaction can encompass one or more individual atomic transactions.

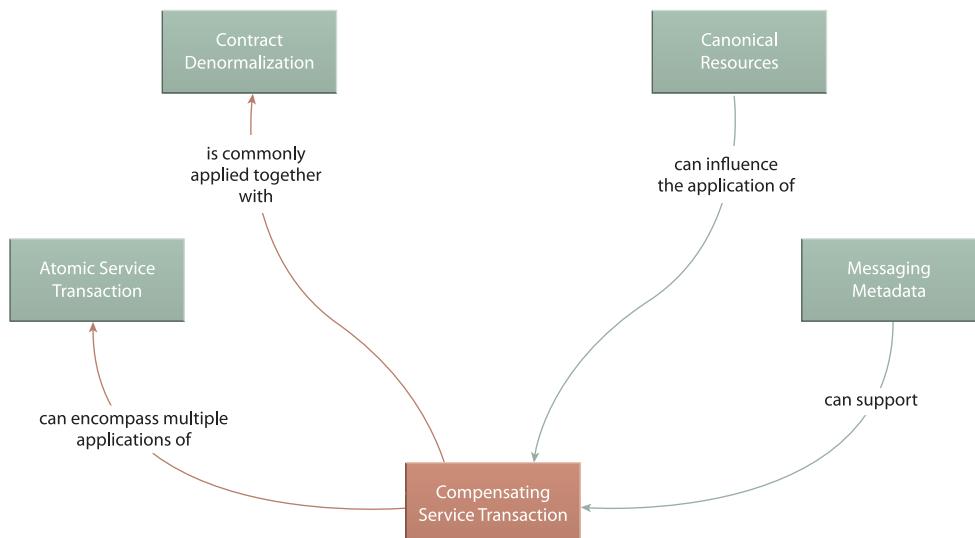


Figure 19.20

Compensating Service Transaction relates to several messaging and composition design patterns.

The ability to define and carry out compensation logic positions Compensating Service Transaction as a core part of Orchestration (701).

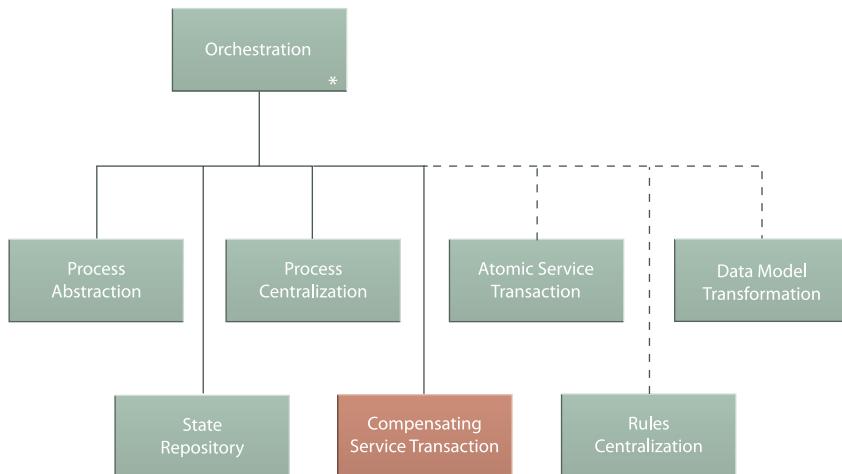


Figure 19.21

Compensating Service Transaction represents a key part of Orchestration (701).

CASE STUDY EXAMPLE

The Inventory Web service developed by Cutit establishes a central access point to their inventory management system, which governs three separate warehouses. This service contains a Transfer operation that allows a consumer to move stock from one warehouse to another. The stock in each warehouse is referred to as an individual inventory.

As shown in Example 19.2, the Transfer operation executes a WS-BPEL routine that carries out a tightly coupled transaction whereby the stock from one inventory (`removeInventory`) must be successfully moved to another (`stockToInventory`). Should the Transfer operation fail, the original state of the stock must be restored in both inventories.

```

<scope name="InventoryMgmt">
  <invoke name="RemoveInventory"
    partnerLink="InventorySystem"
    portType="ns1:InventoryService"
    operation="removeInventory"
    inputVariable="RemoveInventory_Input"
    outputVariable="RemoveInventory_Output" />

```

```
<invoke name="Invoke_StockToInventory"
    partnerLink="InventorySystem"
    portType="ns1:InventoryService"
    operation="stockToInventory"
    inputVariable="newStockInventory_Input"
    outputVariable="newStockInventory_Output"/>
</scope>
```

Example 19.2

This WS-BPEL routine carries out a transaction that must be committed if successful or rolled back in the event of a failure condition.

Transfers can require a great deal of runtime processing, especially when a single transfer consists of a bundle of numerous individual stock items. Due to the nature of the inventory management system API, the Transfer operation is required to repeatedly access the system to perform queries in relation to the source warehouse. It must then carry out another set of queries for the destination warehouse to ensure that there is room for the stock to be transferred. Finally, updates need to be performed, which again requires repeated access.

Throughout all of this interaction, the Inventory service further needs to lock inventory records until the transfer is completed. This has resulted in an unreasonable strain on the Inventory service and the underlying inventory management system, and has further affected other parts of the Cutit enterprise.

As a result, Cutit architects decide to change the functionality of the Transfer operation so that it carries out a compensating transaction. To implement this, the WS-BPEL code is redeveloped to incorporate a compensation handler designed to call an undo operation, as shown here:

```
<scope name="InventoryMgmt">
    <faultHandlers>
        <catch faultName="ns2:NoSpaceAvailableFault">
            <compensate name="Compensate_Withdraw"/>
        </catch>
    </faultHandlers>
    <sequence name="Sequence_1">
        <scope name="WithdrawInventory">
            <compensationHandler>
                <invoke name="undoWithdrawInventory"
                    partnerLink="InventorySystem"
                    portType="ns1:InventoryService"
                    operation="undoWithdrawInventory"
```

```
    inputVariable="undoWithdrawInventory_Input"
    outputVariable="undoWithdrawInventory_Output" />
  </compensationHandler>
  <invoke name="RemoveInventory"
    partnerLink="InventorySystem"
    portType="ns1:InventoryService"
    operation="withdrawInventory"
    inputVariable="WithdrawInventory_Input"
    outputVariable="WithdrawInventory_Output" />
</scope>
<invoke name="Invoke_StockToInventory"
    partnerLink="InventorySystem"
    portType="ns1:InventoryService"
    operation="stockInventory"
    inputVariable="newStockInventory_Input"
    outputVariable="newStockInventory_Output" />
</sequence>
</scope>
```

Example 19.3

A compensation routine that invokes an undo operation (highlighted) in the event that the StockToInventory call fails. The `catch` construct catches the fault condition and then invokes the `undoWithdrawInventory` operation.

Chapter 20



Service Interaction Security Patterns

Data Confidentiality

Data Origin Authentication

Direct Authentication

Brokered Authentication

When designing mission-critical enterprise solutions, which are often comprised of complex service compositions, services can be subjected to a variety of different usage scenarios, each of which can introduce unique security risks and requirements. Designing effective compositions therefore requires that services be prepared for a range of runtime interaction security challenges.

The upcoming Data Confidentiality (641) and Data Origin Authentication (649) patterns focus on applying security at the message level to protect sensitive message data from unintended exposure and tampering. Direct Authentication (656) and Brokered Authentication (661) establish security controls that enable services to verify that only intended consumers will gain access to sensitive message data.

NOTE

The simplified code fragments in the upcoming case study example sections highlight different security technologies that are not explained in this book. Specifically, code associated with the following standards is provided:

- WS-Security
- XML Encryption
- XML Signature
- SAML

To learn more about any of these specifications, visit www.soaspecs.com. For more formal and detailed examples of messages that incorporate encryption and digital signatures in particular, view Appendix D of the *WS-I Sample Application Security Architecture* document (which can also be accessed at SOASpecs.com).

Data Confidentiality

By Jason Hogg, Don Smith, Fred Chong, Tom Hollander, Wojtek Kozaczynski, Larry Brader, Nelly Delgado, Dwayne Taylor, Lonnie Wall, Paul Slater, Sajjad Nasir Imran, Pablo Cibraro, Ward Cunningham



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

Problem	Within service compositions, data is often required to pass through one or more intermediaries. Point-to-point security protocols, such as those frequently used at the transport-layer, may allow messages containing sensitive information to be intercepted and viewed by such intermediaries.
Solution	The message contents are encrypted independently from the transport, ensuring that only intended recipients can access the protected data.
Application	A symmetric or asymmetric encryption and decryption algorithm, such as those specified in the XML-Encryption standard, is applied at the message level.
Impacts	This pattern may add runtime performance overhead associated with the required encryption and decryption of message data. The management of keys can further add to governance burden.
Principles	Service Composability
Architecture	Inventory, Composition, Service

Table 20.1

Profile summary for the Data Confidentiality pattern.

Problem

Message data can flow over insecure networks—either within an organization's internal network or across public networks. A conventional approach is to protect message data at the transport layer by encrypting the connection between a service and its consumer through technologies such as SSL and TLS. These technologies provide point-to-point data protection that hides message data from eavesdroppers between two points in a network.

Transport layer security is effective for many point-to-point data exchanges because the services and consumers involved are pre-defined, and the risk of unwanted exposure to

sensitive data via nontrusted intermediaries tends to be low. However, when messages are exchanged as part of a composition or via message paths with various intermediaries, they can be exposed to the following threats for which point-to-point transport layer security does not provide protection:

- Agents and other services may be able to gain access to message data because while they are in possession of the message data, it is not encrypted.
- Sensitive data might be further vulnerable while temporarily persisted in a message queue, database, or file, and eavesdropper programs located along a network might be capable of gaining access to this data whenever it leaves a secure area (such as a protected memory space) or crosses a communication line that is not encrypted (such as a public network).

Figure 20.1 illustrates where data can be exposed along a message path.

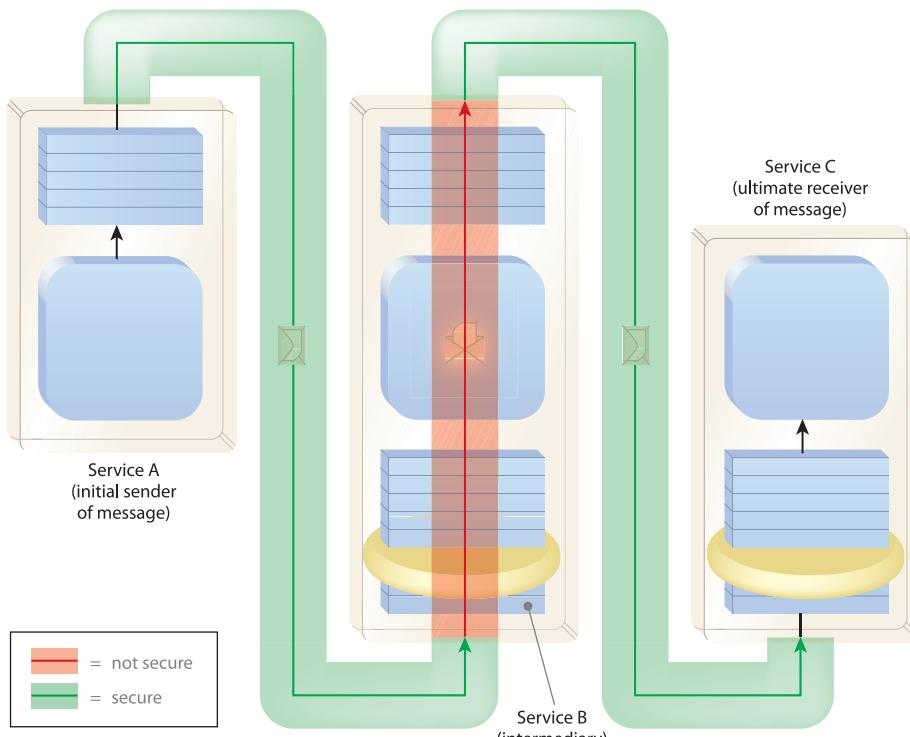


Figure 20.1

Point-to-point or transport layer security measures can only protect a message while in between service transmissions (green zone). It does not protect the message while in the service's possession (red zone).

Solution

To fully protect a message's contents, message layer encryption technologies are applied so that the security of the data is embedded and remains with the message and is only available to authorized recipients (Figure 20.2).

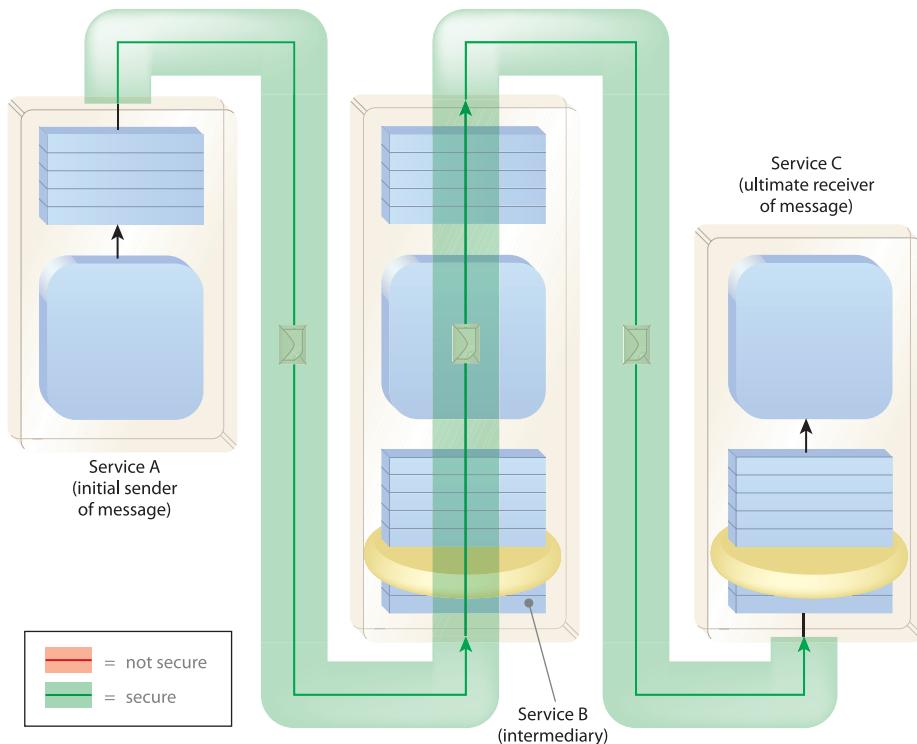


Figure 20.2

Data Confidentiality protects the message while in transit between services and while in the possession of unauthorized intermediaries.

Application

This design pattern is most commonly applied to Web services via the XML-Encryption technology referenced from within the WS-Security standard. XML-Encryption converts unencrypted message data, known as *plaintext*, into encrypted data called *ciphertext*. Plain-text data is encrypted with an algorithm and a cryptographic key. The resulting ciphertext is then converted back to plaintext by the intended message recipient who possesses a key that can decrypt the data.

There are two common types of cryptography that provide data confidentiality: *symmetric* and *asymmetric*. While both follow a similar process, each has its own unique characteristics.

With symmetric cryptography, the sender and recipient share a common key that is used to perform both encryption and decryption. Symmetric cryptography relies on a common secret key and a symmetric encryption algorithm, which transforms data between plaintext and ciphertext. With asymmetric cryptography (also known as public key cryptography), the sender encrypts data with one key, and the recipient uses a different key to decrypt the ciphertext. The encryption key and its matching decryption key are often referred to as a public/private key pair.

In cases where more than one message exchange occurs between a service and consumer, a “high-entropy” shared secret can be negotiated so that the first exchange includes a shared secret that is encrypted and based on the newly generated shared secret; additional message exchanges are performed symmetrically.

Impacts

Cryptographic operations are computationally intensive, which may have an impact on system resource usage. Also the task of managing and safeguarding encryption keys can introduce significant governance overhead depending on the quantity of keys used, the type of encryption chosen, and the overall key management infrastructure.

Furthermore, the security provided by Data Confidentiality is not absolute. Factors that need to be taken into account include the following:

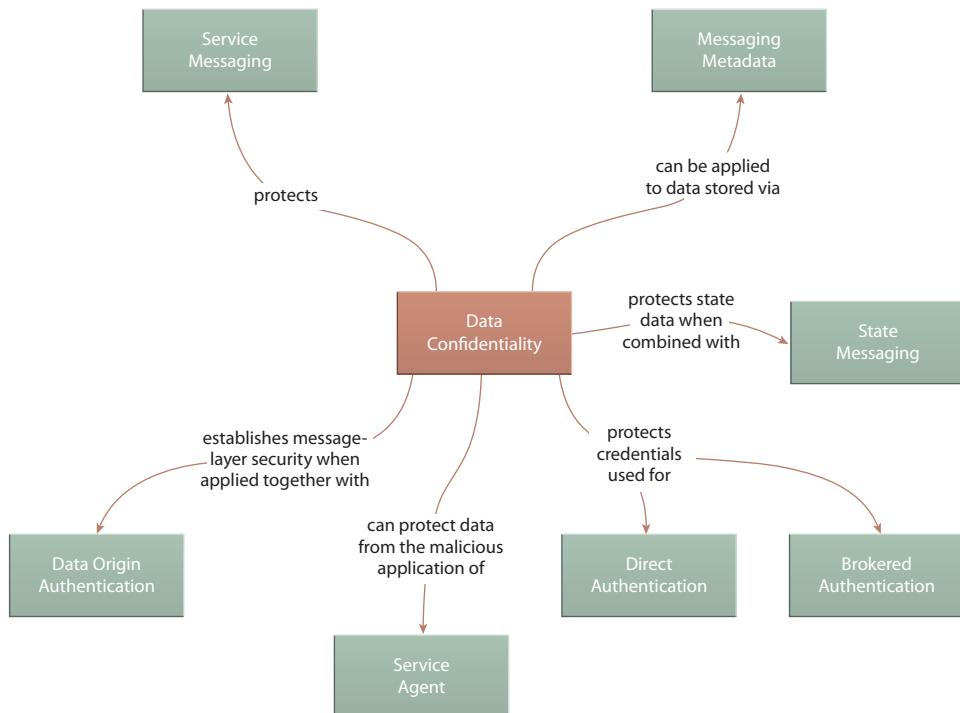
- Encryption does not prevent data tampering. For example, attackers can still replace encrypted data bits in transit, which can cause the message recipient to decrypt the data to something other than the original plaintext. This vulnerability is addressed by Data Origin Authentication (649).

- If too much data is encrypted with the same symmetric key, an attacker can intercept several messages and attempt to cryptographically attack the encrypted messages with the goal of obtaining the symmetric key. To minimize this risk, session-based encryption keys with a relatively short life span are typically used.
- Much of the strength of symmetric encryption algorithms comes from the randomness of their encryption keys. If keys originate from a source that is not sufficiently random, attackers may narrow down the number of possible values for the encryption key.
- Asymmetric encryption requires more processing resources than symmetric encryption. For this reason, asymmetric encryption is usually optimized by adding a one-time, high-entropy symmetric key to encrypt a message and then asymmetrically encrypt the shared key. This reduces the size of the data that is asymmetrically encrypted, which also improves performance.
- Use of encryption algorithms that have not been subjected to rigorous review by trained cryptologists may contain undiscovered flaws that can be exploited by attackers. Therefore, well-known encryption algorithms that have withstood years of rigorous attacks and scrutiny should be used.
- Different countries may recognize different standards for data protection.

Relationships

Because the purpose of Data Confidentiality is to secure message contents, it relates directly to Service Messaging (533) and also Messaging Metadata (538), as encryption can be applied to message headers in addition to the message body content. For example, security credentials, as used by Direct Authentication (656) and Brokered Authentication (661), are often stored in message headers and can therefore be protected via encryption.

This pattern is often combined with Data Origin Authentication (649) to realize complete message layer security so that intermediary processing layers, such as those established by Service Agent (543), cannot be used to gain unauthorized access to message data.

**Figure 20.3**

Data Confidentiality combines messaging and security considerations.

CASE STUDY EXAMPLE

Every company involved with the forestry industry must be registered with the FRC. They are required to pay annual dues to remain in good standing and may also be forced to pay penalties as a result of policy violations.

The processing of payments can be regularly made online via a set of exposed Web services that can issue invoices, collect the payment transfer, and then confirm payments via electronic receipts. To interface with these externally facing services, organizations have the choice of submitting SOAP messages with plaintext or WS-Security-based content. The transmission channel between the organization and the FRC is encrypted via standard SSL.

Cutit Saws does not initially see the value in taking the time to outfit their payment messages with special message layer security. They only issue payments once a year and

have never been fined for any policy violations. They therefore simply send a plaintext message, as follows:

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/" ...>
  <Body>
    <frc:Payment>
      <frc:InvoiceNumber>
        738345
      </frc:InvoiceNumber>
      <frc:Amount>
        $137.14
      </frc:Amount>
      <frc:Date>
        02.06.08
      </frc:Date>
    </frc:Payment>
  </Body>
</Envelope>
```

Example 20.1

A sample SOAP message with no encryption.

Alleywood, on the other hand, has become accustomed to being fined for policy violations on a regular basis. They are therefore required to issue payments to the FRC on a monthly basis.

The service consumer built by Alleywood to interact with the FRC Payment service is fully enabled to leverage their existing Web services security infrastructure. Example 20.2 shows a sample message sent from Alleywood to which Data Confidentiality has been applied:

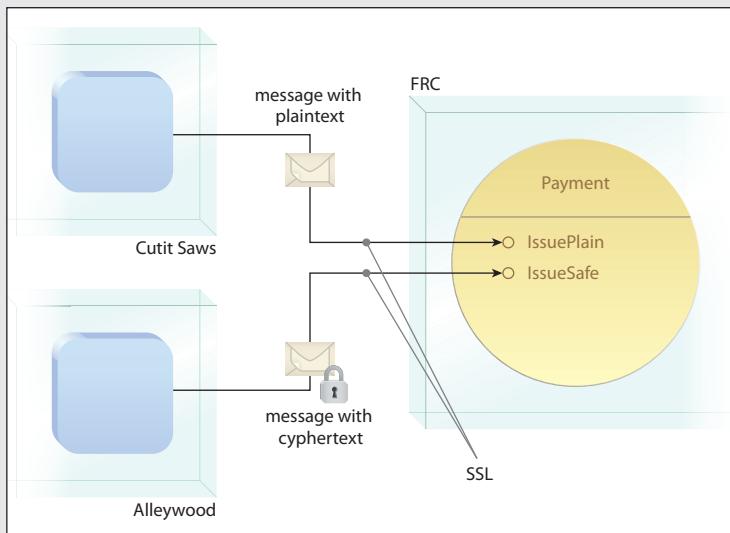
```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/" ...>
  <Body>
    <xenc:EncryptedData xmlns="..." Type="...">
      <xenc:EncryptionMethod Algorithm=
        "http://www.w3.org/2001/04/xmlenc#aes128-cbc" />
      <xenc:CipherData>
        <xenc:CipherValue>
          08FJF0239J
        </xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptionMethod>
```

```
</xenc:EncryptedData>  
</Body>  
</Envelope>
```

Example 20.2

A SOAP message with message layer encryption.

As shown in Figure 20.4, the FRC Payment service exposes separate operations for plain and encrypted messages. In either case, the service ensures that the payment values within the message body are encrypted before the message is further routed throughout the extensive FRC enterprise.

**Figure 20.4**

The transport channels between the FRC Payment service and external service consumer programs are encrypted via SSL, regardless of whether the messages themselves provide further message layer encryption.

Data Origin Authentication

By Jason Hogg, Don Smith, Fred Chong, Tom Hollander, Wojtek Kozaczynski, Larry Brader, Nelly Delgado, Dwayne Taylor, Lonnie Wall, Paul Slater, Sajjad Nasir Imran, Pablo Cibraro, Ward Cunningham



How can a service verify that a message originates from a known sender and that the message has not been tampered with in transit?

Problem	The intermediary processing layers generally required by service compositions can expose sensitive data when security is limited to point-to-point protocols, such as those used with transport-layer security.
Solution	A message can be digitally signed so that the recipient services can verify that it originated from the expected consumer and that it has not been tampered with during transit.
Application	A digital signature algorithm is applied to the message to provide “proof of origin,” allowing sensitive message contents to be protected from tampering. This technology must be supported by both consumer and service.
Impacts	Use of cryptographic techniques can add to performance requirements and the choice of digital signing algorithm can affect the level of security actually achieved.
Principles	Service Composability
Architecture	Composition

Table 20.2

Profile summary for the Data Origin Authentication pattern.

Problem

A message sent by a consumer to a service may need to be processed by one or more intermediaries (routers, firewalls, message queues, and so on). The data contained in the message will ultimately influence the behavior of the service after it is received.

There is a risk that an attacker could manipulate messages in transit in order to maliciously alter service behavior (Figure 20.5). Message manipulation can take the form of data modification within the message or even the substitution of credentials that change the message’s apparent origin, thereby allowing an attacker to impersonate a legitimate consumer.

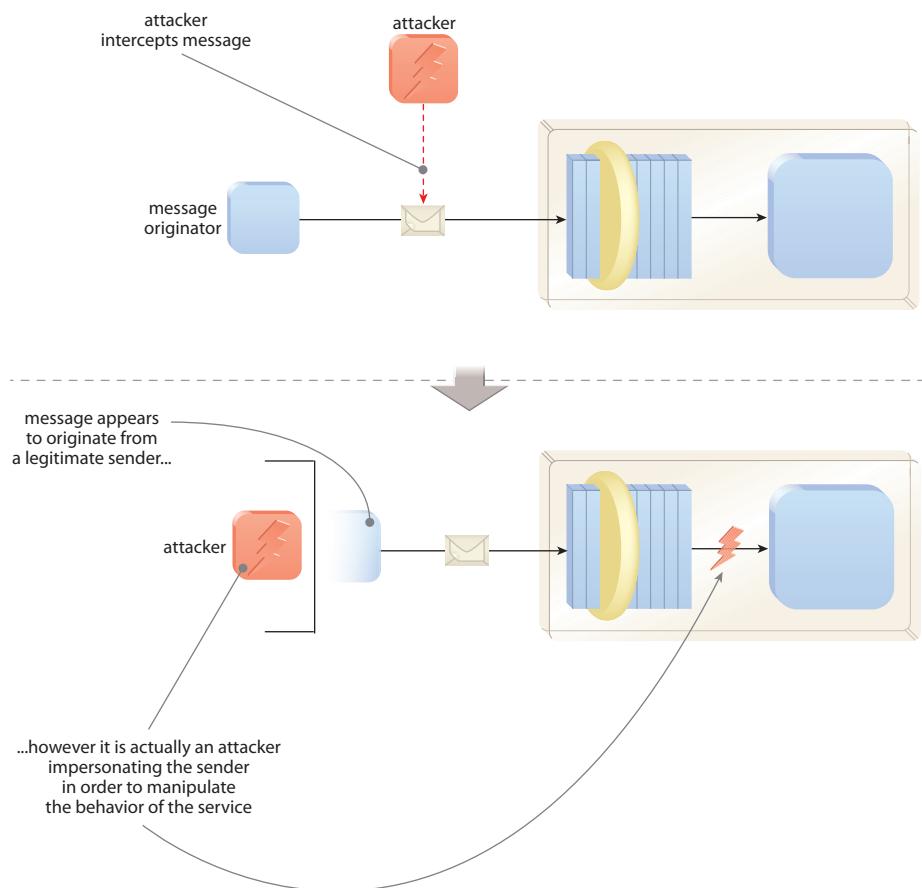


Figure 20.5

An attacker can design or modify an intermediary such that it intercepts a message in order to forward harmful data to a service that views it as a legitimate consumer.

Solution

Digital signature technology is used to enable message recipients to verify that messages have not been tampered with in transit and that they originate from a trusted sender (Figure 20.6).

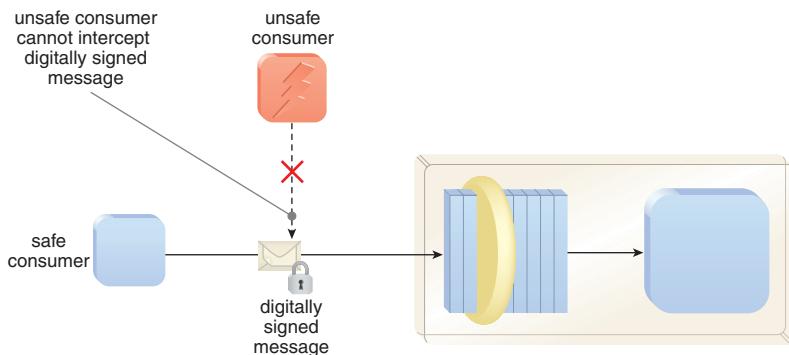


Figure 20.6

In this scenario, the attacker could be attempting to take a valid message and substitute someone else's credentials thereby impersonating the other party, or perhaps the attacker is trying to modify an existing message to the behavior of the service. Either way, when a message is digitally signed, the service can verify the message origin and reject the message if its origin is deemed invalid.

Application

Data Origin Authentication within Web services is typically applied via an XML signature technology from within WS-Security, which enables the recipient of a message to verify that:

- the message has not been altered while in transit (referred to as the message's *data integrity*)
- the message originated from the expected sender (known as the message's *authenticity*)

As with Data Confidentiality (641), this pattern can also be applied with *symmetric* and *asymmetric* variations. Only instead of cryptography, the application of this pattern results in symmetric or asymmetric signatures.

A symmetric signature, commonly known as a Message Authentication Code (MAC), is created by using a shared secret to sign and verify the message. MACs are generated by taking as input a checksum based on the message content and a shared secret. Each MAC can be verified only by a message recipient that has both the shared secret and the original message content that was used to create the MAC.

The most common type of MAC used in Web services is the Hashed Message Authentication Code (HMAC), an algorithm that uses a shared secret and a hashing algorithm to create the signature that is then embedded into the message. The recipient verifies this signature by using the shared secret and the message content to recreate the HMAC and by comparing it to the actual HMAC that was sent in the message.

Asymmetric signatures, on the other hand, are processed with two different keys: One (the private key) is used for creating the signature and another (the public key) for verifying it. These keys are related and are commonly referred to as a *public/private key pair*. The public key is generally distributed with the message, whereas the private key is kept secret by the owner (and is never sent in a message). A signature that is created and verified with an asymmetric public/private key pair is referred to as a *digital signature*.

For both signing and encryption purposes, asymmetric keys are often managed through a Public Key Infrastructure (PKI). Information that describes the consumer is bound to its public key through endorsement from a trusted party to form a certificate that allows a message recipient to verify the private key in a received message signature to the public key in the sender's certificate.

NOTE

The XML-Signature technology can be used to provide symmetric or asymmetric algorithms to SOAP message content. Additional digital signature technologies are available that explicitly use an asymmetric algorithm. These technologies may or may not be part of a SOAP message security architecture.

XML-Signature is part of the WS-Security framework. For more information, visit SOASpecs.com.

It is also worth noting that asymmetric digital signatures can be used to support requirements for non-repudiation. This is because access to the private key is usually restricted to the owner of the key, which makes it easier to verify proof-of-ownership, a value that a client presents to demonstrate knowledge of either a shared secret or a private key to support client authentication. In cases where the consumer denies having performed the action, digital signatures can provide evidence to the contrary. Although digital signatures can help prove non-repudiation, their use may not be sufficient to provide actual legal proof.

Symmetric signatures, on the other hand, cannot support non-repudiation because shared secrets are known by multiple parties. This makes it more difficult to prove that a specific party used the shared secret to sign the message.

Impacts

This pattern shares similar impacts as Data Confidentiality (641) because it too utilizes keys and is subject to the same performance and governance related consequences and also the same types of security risks.

Relationships

Because of its close relationship with achieving message layer security, Data Origin Authentication has almost the identical set of relationships as Data Confidentiality (641).

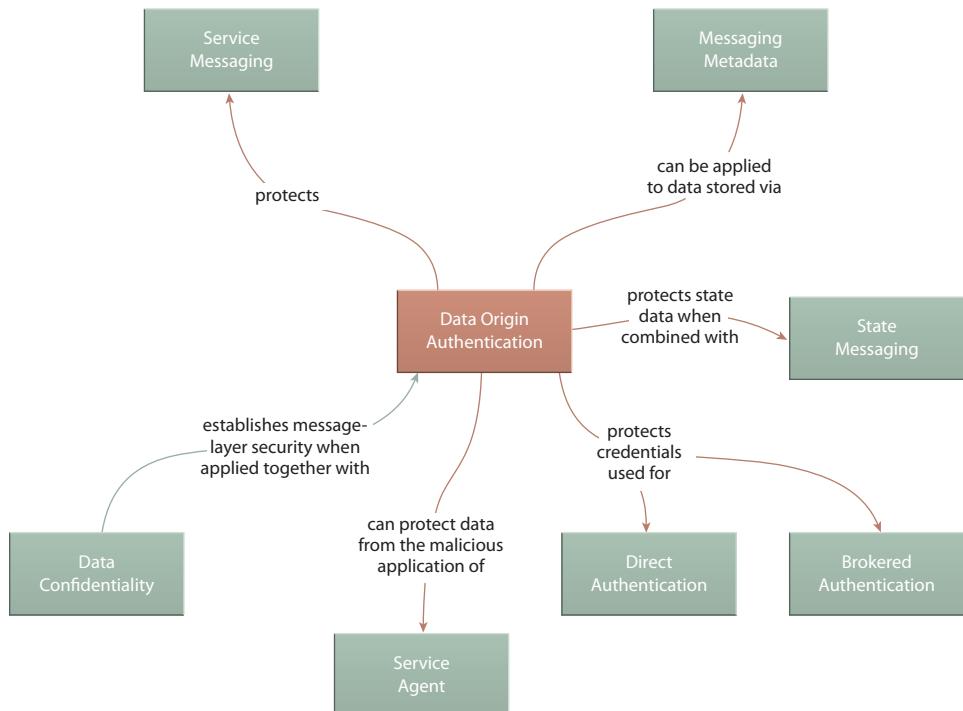


Figure 20.7

As with Data Confidentiality (641), this pattern is focused on applying security to the message and therefore shares the same pattern relationships.

CASE STUDY EXAMPLE

A couple of weeks after the FRC Payment service (introduced in the Data Confidentiality (641) example) went live, an FRC administrator performing a routine review of system logs notices something unusual. For the past three days, the Payment service has periodically redirected messages received from external organizations to an internal dead letter queue even though no exceptions seem to have triggered this. Upon further investigation, the administrator locates the redirected messages and notices that they do not contain valid payment submissions. Instead, they are populated with cryptic content that can't be deciphered by humans.

The administrator immediately calls in the head of FRC's security department. Subsequent to an emergency review, the security manager orders the Payment service shut down. Following some further analysis by the security team, it appears as though the Payment service has been subjected to periodic attacks by Cutit Saws. The Cutit IT manager is quickly contacted, and after some further joint analysis, it is discovered that although the messages indicate that they originated from Cutit Saws, they were, in fact, sent by an malicious consumer program that was impersonating the Cutit consumer. Because Cutit has been sending their messages in plaintext, they had been easily intercepted and modified.

This revelation leads to a new policy that is brought into effect within the next week requiring that all external organizations digitally sign their messages when accessing the Payment service. The Cutit team that was originally hesitant to invest in building security into their message transmissions is now happy to comply, still feeling a bit shook up by this security breach.

Cutit applies Data Origin Authentication, enabling the FRC Payment service to verify the origin of their messages. Should an attacker now try to modify message contents, the manipulations will be detected and the messages rejected.

The following example provides some insight into the new SOAP header that is added to Cutit messages:

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/" ...>
  <Header>
    <wsse:Security>
      <ds:Signature Id="23rml23r0"
        xmlns="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm=
            "http://www.w3.org/TR/2001/
             REC-xml-c14n-20010315"/>
          <ds:SignatureMethod Algorithm=
            "http://www.w3.org/2000/09/
             xmldsig#dsa-sha1"/>
        <ds:Reference
          URI="http://www.w3.org/TR/2000/
            REC-xhtml1-20000126/">
        <ds:DigestMethod Algorithm=
          "http://www.w3.org/2000/09/xmldsig#sha1"/>
        <ds:DigestValue>
          83KFHFR923JLS9
        </ds:DigestValue>
      </ds:SignedInfo>
      <ds:Signature xmlns="http://www.w3.org/2000/09/xmldsig#" Id="23rml23r0">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm=
            "http://www.w3.org/TR/2001/
             REC-xml-c14n-20010315"/>
          <ds:SignatureMethod Algorithm=
            "http://www.w3.org/2000/09/
             xmldsig#rsa-sha1"/>
        <ds:Reference
          URI="http://www.w3.org/TR/2000/
            REC-xhtml1-20000126/">
        <ds:DigestMethod Algorithm=
          "http://www.w3.org/2000/09/xmldsig#sha1"/>
        <ds:DigestValue>
          83KFHFR923JLS9
        </ds:DigestValue>
      </ds:SignedInfo>
      <ds:Signature xmlns="http://www.w3.org/2000/09/xmldsig#" Id="23rml23r0">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm=
            "http://www.w3.org/TR/2001/
             REC-xml-c14n-20010315"/>
          <ds:SignatureMethod Algorithm=
            "http://www.w3.org/2000/09/
             xmldsig#rsa-sha1"/>
        <ds:Reference
          URI="http://www.w3.org/TR/2000/
            REC-xhtml1-20000126/">
        <ds:DigestMethod Algorithm=
          "http://www.w3.org/2000/09/xmldsig#sha1"/>
        <ds:DigestValue>
          83KFHFR923JLS9
        </ds:DigestValue>
      </ds:SignedInfo>
    </ds:Signature>
  </Header>
</Envelope>
```

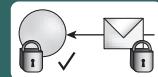
```
        </ds:DigestValue>
    </ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>
    KVMV034045IGKF-R
</ds:SignatureValue>
<ds:KeyInfo>
    ...
</ds:KeyInfo>
</ds:Signature>
</wsse:Security>
</Header>
...
</Envelope>
```

Example 20.3

The FRC now requires that all external organizations issue digitally signed messages when issuing payment data. This example provides a snippet of a SOAP header containing a digital signature.

Direct Authentication

By Jason Hogg, Don Smith, Fred Chong, Tom Hollander, Wojtek Kozaczynski, Larry Brader, Nelly Delgado, Dwayne Taylor, Lonnie Wall, Paul Slater, Sajjad Nasir Imran, Pablo Cibraro, Ward Cunningham



How can a service verify the credentials provided by a consumer?

Problem	Some of the capabilities offered by a service may be intended for specific groups of consumers or may involve the transmission of sensitive data. Attackers that access this data could use it to compromise the service or the IT enterprise itself.
Solution	Service capabilities require that consumers provide credentials that can be authenticated against an identity store.
Application	The service implementation is provided access to an identity store, allowing it to authenticate the consumer directly.
Impacts	Consumers must provide credentials compatible with the service's authentication logic. This pattern may lead to multiple identity stores, resulting in extra governance burden.
Principles	Service Composability
Architecture	Composition, Service

Table 20.3

Profile summary for the Direct Authentication pattern.

Problem

Services are commonly required to handle sensitive or private data that cannot be made available to all potential consumer programs. Furthermore, certain service capabilities may carry out internal processing that should only be triggered by certain types of consumers. Making these capabilities openly available to any consumer will jeopardize the security of the service and any resources it may access or have access to (Figure 20.8).

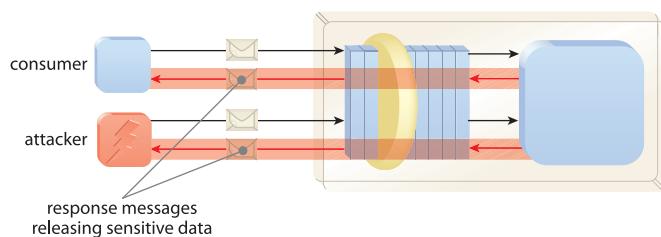


Figure 20.8

When a service is openly accessible, intended consumers and attackers can gain access to potentially sensitive data and logic.

Solution

The service requires the consumer to present credentials for authentication so that additional controls, such as authorization and auditing, can be implemented. The credentials that the consumer presents to the service must include a unique identifier and a shared secret in order for the service to perform authentication. An example of a unique identifier is a username and a shared secret can be a password or a password alternative.

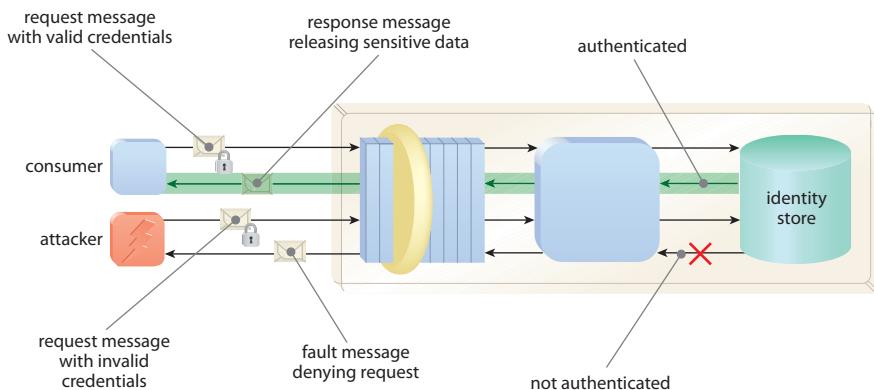


Figure 20.9

By having the service authenticate consumer requests against an identity store, only safe consumers can access sensitive data and logic.

Application

With Direct Authentication, the service is designed to take “direct” responsibility for authentication. As depicted in Figure 20.9, the application of this pattern usually results in a process whereby the service consumer accesses the service and provides credentials that are then authenticated by the service via an identity store that holds the credentials of valid consumers for a particular identity domain. The service verifies that the credentials represent a unique consumer and that it has established “proof-of-possession” of a shared secret. If authentication is successful, the requested data is sent to the consumer, and if not, a fault message is sent to the consumer.

Note that proof-of-possession can be established by the consumer when it provides the actual shared secret to the service or by supplying a password equivalent. One common

password equivalent is a password created using a keyed-hash message authentication (HMAC) function, which uses a checksum calculated from the message and the shared secret in order to generate a password alternative. In order for the service to verify the integrity and authenticity of the message, it retrieves the user's password from the identity store, calculates the checksum of the message, and then uses the HMAC function to recreate the password alternative. If it matches that provided by the consumer, then both the integrity and authenticity of the message has been verified. For the service to authenticate credentials, it must have direct access to the identity store, including appropriate permissions for accessing identity information.

The data stores required by this pattern can be centralized or decentralized. The former model establishes a remote store that is shared by multiple services. This can lead to runtime latency and behavioral fluctuations, depending on usage volume. A decentralized model enables some or all services to own their own individual identity stores. Although this increases autonomy, it can add to the governance burden of keeping multiple identity stores constantly synchronized.

Impacts

Consumers and services must trust each other to manage keys securely. If either mismanage the keys, the service can be repeatedly and unknowingly attacked.

A benefit to Direct Authentication is that if the shared secret is compromised, then only the relationship between one service composition and one consumer is jeopardized. Other compositions using alternative identity stores may not be affected as long as each consumer retains unique credentials. However, because Direct Authentication on its own does not provide single sign-on functionality, consumers will often need to be repeatedly authenticated across services within a service composition. Although this can be avoided by caching the consumer password, it is still not recommended.

Additionally, a malicious consumer can attempt to impersonate a safe consumer by intercepting the transmission of the shared secret. It therefore is advisable that the transport of messages containing credentials be secured.

Also at risk is the identity store itself. Comprehensive security measures must be applied to ensure that its collection of passwords is not breached. This is especially a concern when regular database products are customized to act as repositories for security credentials.

Relationships

As explained in the *Relationships* sections for Data Confidentiality (641) and Data Origin Authentication (649), messages that are exchanged with security credentials can be protected by message layer security controls. These credentials are used to support Direct Authentication.

This pattern relies on the cross-service standardization achieved by Canonical Resources (237) so that composed services that authenticate each other use a common security architecture. In this type of environment, Direct Authentication is usually chosen over Brokered Authentication (661).

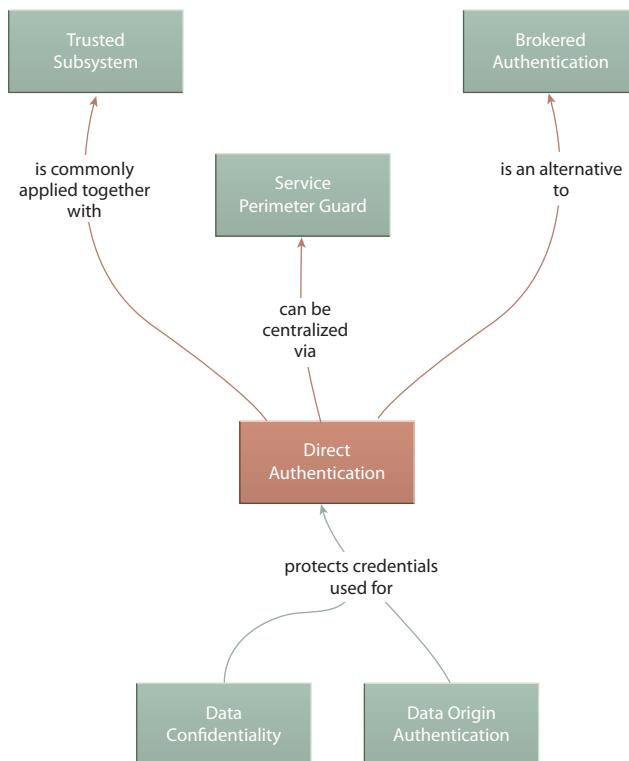


Figure 20.10

Direct Authentication is commonly applied together with other security patterns.

CASE STUDY EXAMPLE

For the previously introduced FRC Payment service to forward messages to their next, destinations, the service needs to supplement them with security credentials for internal authentication. The WS-Security framework has been standardized within the FRC environment and is used for this purpose.

The following simplified example shows a message extract containing metadata in the Header construct that expresses authentication credentials:

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/" ...>
  <Header>
    <wsse:Security>
      <wsse:UsernameToken>
        <wsse:Username>
          sam009ev
        </wsse:Username>
        <wsse:Password Type="wsse:PasswordDigest">
          idkxm2039841sdf
        </wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </Header>
  <Body>
    ...
  </Body>
</Envelope>
```

Example 20.4

The FRC uses WS-Security metadata in headers for the internal transmission of security credentials via SOAP messages.

Brokered Authentication

By Jason Hogg, Don Smith, Fred Chong, Tom Hollander, Wojtek Kozaczynski, Larry Brader, Nelly Delgado, Dwayne Taylor, Lonnie Wall, Paul Slater, Sajjad Nasir Imran, Pablo Cibraro, Ward Cunningham



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?

Problem	Requiring the use of Direct Authentication (656) can be impractical or even impossible when consumers and services do not trust each other or when consumers are required to access multiple services as part of the same runtime activity.
Solution	An authentication broker with a centralized identity store assumes the responsibility for authenticating the consumer and issuing a token that the consumer can use to access the service.
Application	An authentication broker product introduced into the inventory architecture carries out the intermediary authentication and issuance of temporary credentials using technologies such as X.509 certificates or Kerberos, SAML, or SecPAL tokens.
Impacts	This pattern can establish a potential single point of failure and a central breach point that, if compromised, could jeopardize an entire service inventory.
Principles	Service Composability
Architecture	Inventory, Composition, Service

Table 20.4

Profile summary for the Brokered Authentication pattern.

Problem

Services can have a wide variety of consumers, many of which will be unknown when the service is first designed (Figure 20.11). Establishing trust directly between a consumer and service, as per Direct Authentication (656), often requires out-of-band communication that can hinder consumers and services from interacting dynamically.

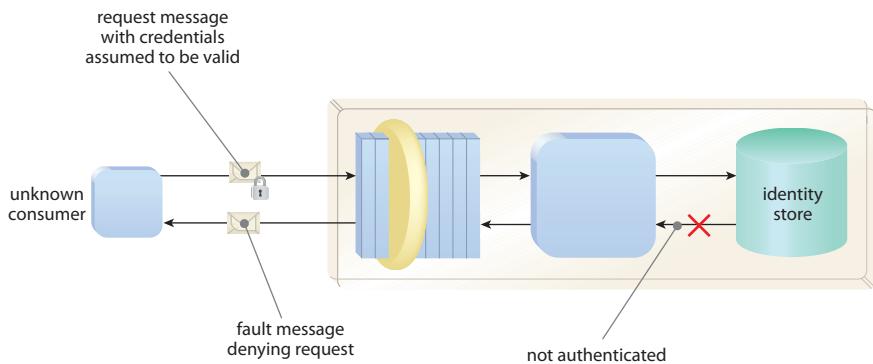


Figure 20.11

A consumer attempts to access a service with credentials that turn out to be incorrect.

Furthermore, even when Direct Authentication (656) is successfully applied, if a given consumer requires access to numerous services as part of the same runtime service activity, it may be asked to provide separate credentials for each service (Figure 20.12). In this case, the consumer may need to cache credentials for use with additional runtime activities, which may result in the consumer having to persist a password temporarily, which introduces additional security threats.

Solution

An authentication broker is added as an architectural extension capable of validating consumer credentials without the need for consumers to have direct relationships with the services they need to access. Both consumers and services trust the authentication broker, allowing it to become a centralized authentication and credential issuing mechanism within the overall inventory architecture (see Figure 20.13).

This centralized security platform can simplify the development of individual services and can further reduce the governance burden associated with identity management.

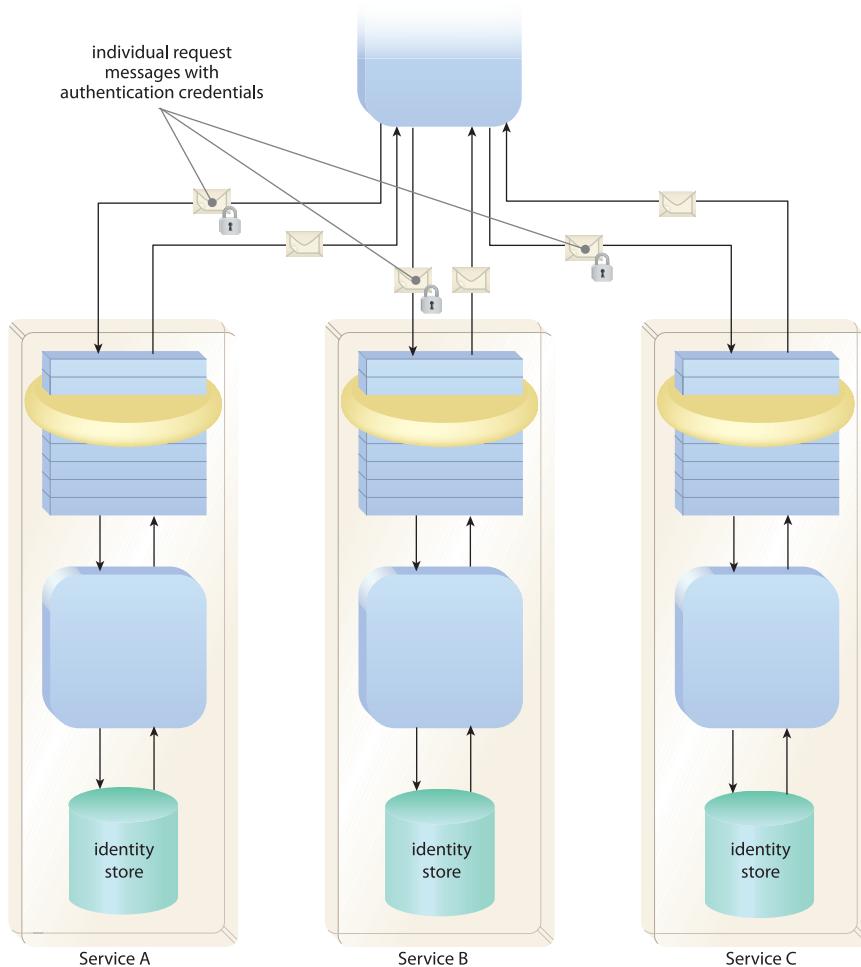


Figure 20.12

Using Direct Authentication (656), a consumer is forced to send credentials to and get authenticated by each service it composes.

Application

This pattern is generally applied by the deployment of an authentication broker product or platform. The internal architectures of these products can vary in how they carry out the broker functions. However, for the most part, they follow the process illustrated in Figure 20.13 (note the use of the small lock symbol that represents a token).

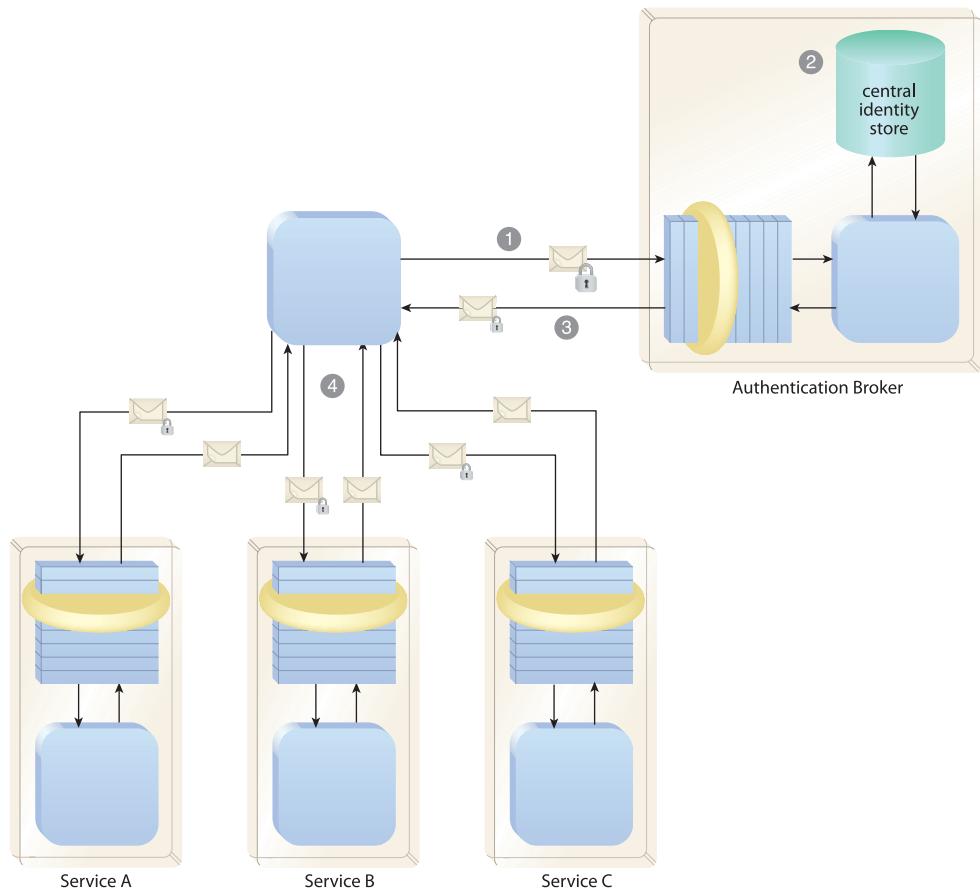


Figure 20.13

The consumer submits a request with credentials to the authentication broker (1), which the broker authenticates against a central identity store (2). The broker then responds with a token (3) that the consumer can use to access Services A, B, and C (4), none of which require their own identity store.

Figure 20.13 demonstrates how authentication brokers can be capable of issuing tokens to consumers that are used for future interactions. For example, if the consumer requests access to a specific service, the broker can issue a token to an authorized consumer that is specifically scoped for that service, potentially allowing the consumer to repeatedly interact with the service using the same token issued by the broker.

There are a variety of security technologies associated with authentication brokers:

- the X.509 PKI infrastructure that uses a Certificate Authority for signing and issuing X.509 certificates

- the Kerberos protocol, which uses an Authentication service and a Ticket Granting service for issuing Kerberos tickets
- the WS-Trust specification, which describes a protocol used by Security Token Services (STS) for issuing tokens such as SAML and SecPAL

Claims held in security tokens can contain sensitive data and must therefore be protected in transit via Data Confidentiality (641) or the use of a transport-layer security, such as SSL. Security tokens must be signed by the issuing authentication broker. If they are not, their integrity cannot be verified, which could result in attackers trying to issue false tokens. Furthermore, without the security token being signed, there would be no way for a service to verify that the token was issued by a trusted authentication broker.

Note also that trust relationships can be established between different authentication brokers. In other words, one broker can issue security tokens that are used across organizational boundaries and autonomous security domains, each of which can have their own broker.

Impacts

The centralized trust model that Brokered Authentication uses could create a single point of failure. Some types of authentication brokers, such as the Kerberos Key Distribution Center (KDC), must be constantly online and available to issue security tokens to consumers. Should such an authentication broker become unavailable at any point, it could cripple service communication. This can be mitigated somewhat via Redundant Implementation (345) or by implementing backup authentication brokers.

Additional tradeoffs associated with the validity periods of issued tokens also need to be considered. For example, the Kerberos protocol typically issues tokens for eight hours, whereas X.509 certificates will often be issued for much longer validity periods (often around a year), in which case sophisticated revocation systems, such as those provided by Certificate Revocation Lists (CRLs) and the Online Certificate Status Protocol (OCSP), are required.

Furthermore, should an authentication broker ever be compromised, attackers may gain access to confidential identify information or even the ability to falsely issue security tokens, enabling them to gain access to and perform malicious activities against many services.

Relationships

Brokered Authentication is an alternative to Direct Authentication (656) but shares most of the same pattern relationships. The broker functionality provided by this pattern is often conveniently centralized via Service Perimeter Guard (394).

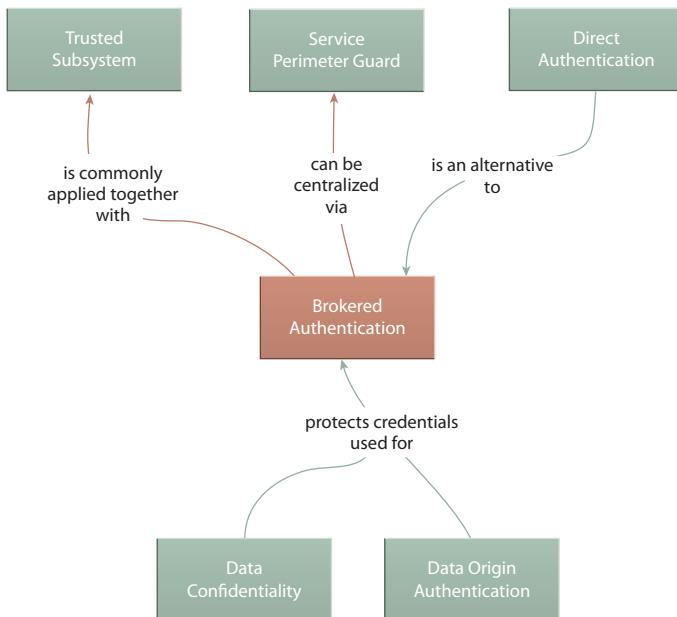


Figure 20.14

Brokered Authentication shares the same relationships as Direct Authentication (656) but is also commonly applied together with Service Perimeter Guard (394).

CASE STUDY EXAMPLE

The FRC Payment service explained in the preceding examples is required to forward collected payment information to the following four services:

- Audit service
- Logging service
- Accounts Receivable service
- Notification service

The first three require that the Payment service (acting as the consumer) be authenticated. The initial Direct Authentication (656) design FRC architects used required that each of these three services separately authenticate the Payment service. After a while, it became evident that a few issues made this architecture unacceptable:

- Due to governance reasons, each service ended up with its own identity store.

- It became increasingly difficult to keep the identity stores in synch.
- Runtime latency was introduced as a result of having to repeatedly re-authenticate a consumer multiple times.

After explaining these issues in a business case for their manager, the FRC architects gained funding for the purchase of an authentication broker.

This product uses a Security Token Service (STS) in its perimeter network that serves the purpose of authenticating users from remote consumer locations, and then uses a SAML token signed by the STS. The issuance of these tokens is managed automatically through trust policies that specify the services supported by the broker. The broker is further configured to require that the Payment service authenticate to the STS using X.509 certificates issued by a separate Certificate Authority (CA). The trust policies on the STS specify which CAs are allowed to issue certificates and attributes from certificates that should be mapped from the certificate to the SAML token that is issued.

The following example shows a snippet of the SAML code used in a token:

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/" ...>
  <Header>
    <wsse:Security>
      <saml:Assertion ...>
      <saml:Conditions ...>
      <saml:AuthorizationDecisionStatement>
        <saml:Actions>
          ...
          <saml:Action>
            Access Audit
          </saml:Action>
        </saml:Actions>
        ...
      </saml:AuthorizationDecisionStatement>
    </wsse:Security>
  </Header>
  <Body>
    ...
  </Body>
</Envelope>
```

Example 20.5

A SOAP message containing a SAML token.

This page intentionally left blank

Chapter 21



Transformation Patterns

Data Model Transformation

Data Format Transformation

Protocol Bridging

When introducing service-oriented architectures into legacy environments or when delivering services via bottom-up approaches, it is not uncommon to encounter interoperability challenges that can only be overcome by programs with brokerage features.

These three common patterns have been part of integration architectures for many years and continue to be valuable and even essential in modern-day SOA implementations. Each addresses an aspect of data exchange:

- *Communications Protocol* – Software programs can be built to communicate with different technologies or different versions of the same technology. Either way, when disparity between wire-level protocols exist, some form of intermediary program is required to “bridge” these differences.
- *Data Format* – Even if the underlying communications protocol is compatible, two programs that need to exchange data may offer this data via different character sets, file formats, or data formats (such as XML and CSV). For this, an additional type of intermediary processing logic is required to convert one format to the other.
- *Data Model* – Even when the communications protocol and data formats are compatible, for software programs to exchange any form of structured data (such as business documents), the definition of the model that the data structure is based on (the data model) can be different at transmitting and receiving ends. This leads to a requirement for the runtime transformation of information from one data model into another.

Protocol Bridging (687), Data Format Transformation (681), and Data Model Transformation (671) respectively address these three interoperability obstacles. Although their methods are well-established and proven, the required use of these patterns can indicate an inability to achieve the standardization required to realize service-orientation to its full extent within a given environment.

Each pattern introduces a layer of intermediary processing that has similar impacts on performance, design complexity, and development effort. The overuse of these patterns is therefore considered an anti-pattern.

Data Model Transformation

How can services interoperate when using different data models for the same type of data?



Problem	Services may use incompatible schemas to represent the same data, hindering service interaction and composition.
Solution	A data transformation technology can be incorporated to convert data between disparate schema structures.
Application	Mapping logic needs to be developed and deployed so that data compliant to one data model can be dynamically converted to comply to a different data model.
Impacts	Data model transformation introduces development effort, design complexity, and runtime performance overhead, and overuse of this pattern can seriously inhibit service recomposition potential.
Principles	Standardized Service Contract, Service Reusability, Service Composability
Architecture	Inventory, Composition

Table 21.1

Profile summary for the Data Model Transformation pattern.

Problem

For any service-enabled part of an enterprise where Canonical Schema (158) has not been successfully applied, services delivered as part of different projects run a high risk of representing similar data using different schemas (Figure 21.1).

This will typically not affect immediate service implementations, as the schemas may very well have been standardized across the initial solution or composition. However, when these same services need to be recomposed, schema incompatibilities can impose significant interoperability challenges.

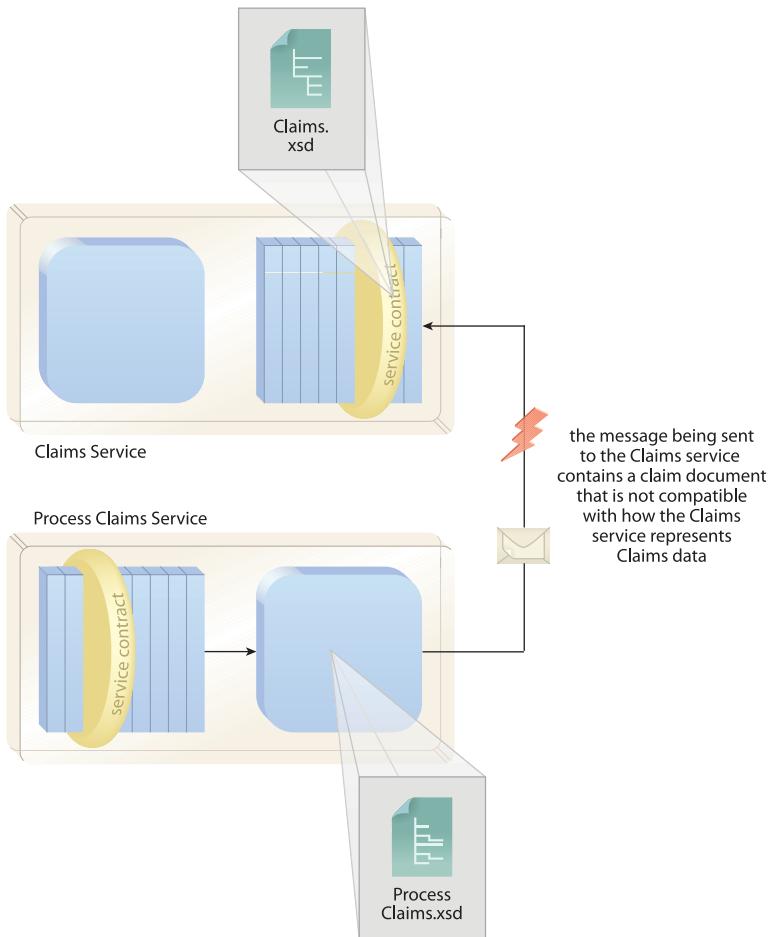


Figure 21.1

The Process Claims service represents a claim record using a different schema than the one required by the Claims service contract. This incompatibility prevents the cross-service exchange of a claims document.

Solution

Data model transformation logic can be introduced to carry out the runtime conversion of data, so that data complying to one data model can be restructured to comply to a different data model (Figure 21.2). This extends a non-standardized messaging framework, enabling it to dynamically overcome disparity between the schemas used by a service contract and messages transmitted to that contract.

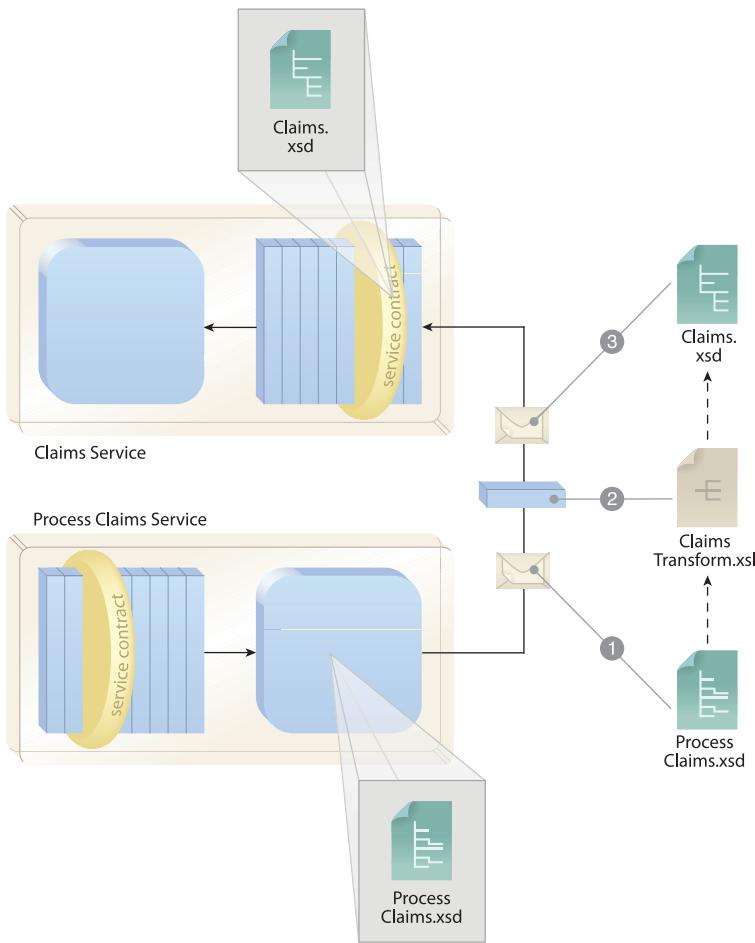


Figure 21.2

An XSLT style sheet containing data model mapping logic (2) is added as a form of intermediary processing that is executed at runtime. With each transmission, the data model of the claims document is converted from the schema used by the Process Claims service (1) to the data model compliant with the schema used by the Claims service (3). This runtime transformation logic can reside with either service architecture or as part of a separate middleware platform.

Application

Formalized data transformation is an established concept that dates back to the EAI era where data model incompatibilities were often resolved via broker services that existed as part of middleware platforms.

When services are implemented as Web services, XSLT is generally used to define the mapping logic that is subsequently executed to perform the transformation at runtime. In fact, the use of XSLT style sheets represents the most common application of this pattern.

Although Data Model Transformation is a fundamental and essential part of most service-oriented architectures, it represents a pattern that is used only out of necessity. Many of the service contract patterns described in this book, in fact, result in a reduction or even an elimination of data transformation requirements. However, the realities of legacy encapsulation, the application of Domain Inventory (123), and common governance challenges usually introduce schema disparity that still must be dealt with.

This pattern solves an important problem, but it is equally important to be constantly aware of the fact that it is only applied when other patterns cannot be realized to their full potential. Due to the constant emphasis on “transformation avoidance” throughout service-orientation, it is recommended to consider this pattern as a last resort to solving interoperability problems.

NOTE

Data Model Transformation further addresses the transformation required between different versions of the same data model. For XML schemas, XSLT can also be used for this purpose.

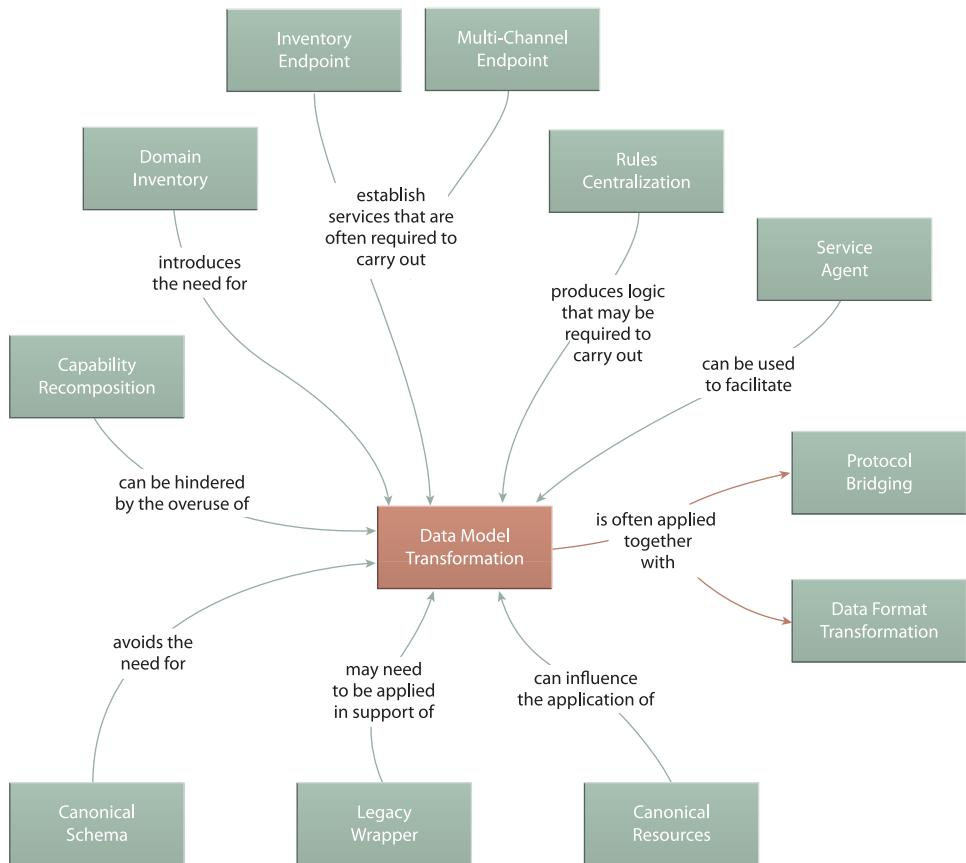
Impacts

The use of this pattern has several consequences:

- The development of the mapping logic adds time and effort to the creation of service compositions.
- The incorporation of data model transformation logic introduces design complexity into a service composition and the service inventory as a whole.
- Data model transformation logic introduces a runtime layer wherein the execution of mapping logic adds performance overhead every time services with disparate schemas need to exchange data.

Relationships

Canonical Schema (158) has a well-known “tug-of-war” relationship with Data Model Transformation, in that this pattern generally needs to be applied to whatever extent Canonical Schema (158) is not realized.

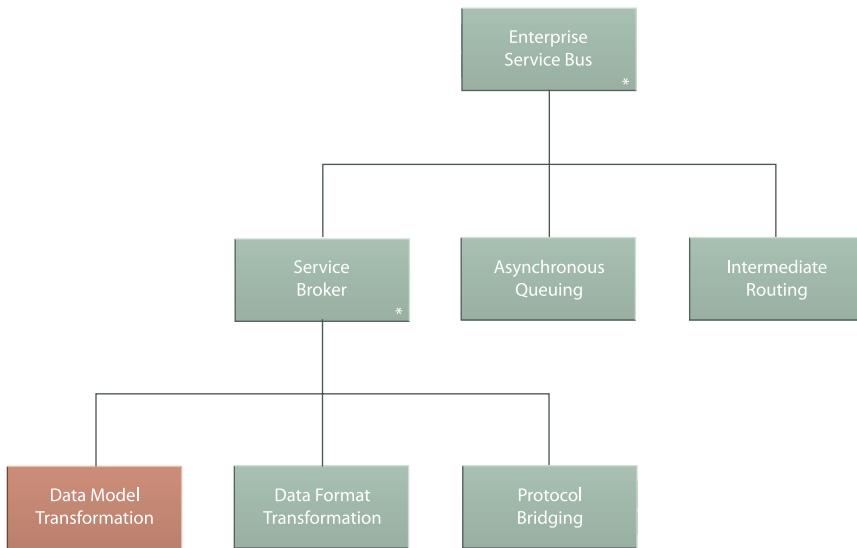
**Figure 21.3**

Perhaps the most common broker pattern used in composition architectures, Data Model Transformation has many ties to a diverse collection of patterns that are affected by its application.

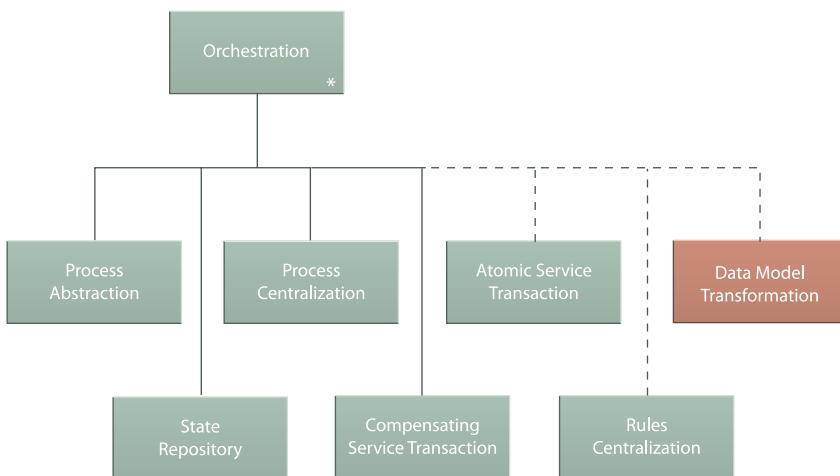
When in use, it can help overcome disparity between services that may be introduced by patterns like Domain Inventory (123) and Inventory Endpoint (260). It also addresses conversion requirements that may result from Legacy Wrapper (441) and Rules Centralization (216).

Although it provides important functionality, its overuse is an anti-pattern that can undermine the goals of Capability Recomposition (526).

As shown in Figures 21.4 and 21.5, Data Model Transformation is a core part of Service Broker (707) and Enterprise Service Bus (704) and also the only broker-related pattern associated as an optional extension of Orchestration (701).

**Figure 21.4**

Data Model Transformation is one of three transformation-related design patterns that comprise Service Broker (707) and Enterprise Service Bus (704).

**Figure 21.5**

Data Model Transformation is also one of the optional patterns associated with Orchestration (701).

CASE STUDY EXAMPLE

A year before they were acquired, one of Alleywood's largest clients established an online business exchange system that required Alleywood to receive purchase orders and produce invoices based on the Universal Business Language (UBL) schema. Since that time, Alleywood has continued to use UBL for a variety of accounting documents. Tri-Fold, on the other hand, has always used its own proprietary XML Schema definition for these same types of documents.

Now that Alleywood and Tri-Fold have developed their respective domain service inventories, internal interoperability requirements begin to emerge leading to the need for Alleywood and Tri-Fold services to exchange accounting data. The first document that needs to be addressed is the purchase order.

A fragment of Tri-Fold's XML Schema definition for a purchase order is shown here:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tri="..." targetNamespace="..."
  elementFormDefault="qualified">
  <xsd:element name="order-form" type="tri:order-form.type"/>
  <xsd:element name="customer" type="tri:customer.type"/>
  <xsd:element name="delivery" type="tri:delivery.type"/>
  <xsd:element name="goods" type="tri:goods.type"/>
  <xsd:complexType name="order-form.type">
    <xsd:sequence>
      <xsd:element ref="tri:customer"/>
      <xsd:element ref="tri:delivery"/>
      <xsd:element ref="tri:goods"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string"
      use="required"/>
    <xsd:attribute name="timestamp" type="xsd:dateTime"
      use="required"/>
  </xsd:complexType>
  ...
</xsd:schema>
```

Example 21.1

Code from the Tri-Fold purchase order XML Schema definition.

A sample purchase order document instance of this schema would look something like this:

```
<order-form xmlns="..." id="20080212-AA"
  timestamp="2008-02-12T13:20:54-06:00">
  <customer ref="alleywood">
    <name>Alleywood Lumber Company</name>
    <street>128 Wood Avenue</street>
    <city zip="60601" state="IL">Chicago</city>
    <contact phone="+1 555 55 55" name="John Smith"/>
  </customer>
  <delivery date="2008-02-24+P1D">
    <street>42 Forest Street</street>
    <city zip="60601" state="IL">Chicago</city>
  </delivery>
  <goods quantity="5" ref="sp-457">
    <amount total="5412.45" unit="1082.49"/>
    <note>Special Paper</note>
  </goods>
</order-form>
```

Example 21.2

The beginning of a Tri-Fold purchase order document that corresponds to the schema from Example 21.1.

In contrast, Alleywood's UBL-based purchase order document is twice the size. Here is an excerpt from a document instance comparable to Tri-Fold's:

```
<Order xmlns="..." xmlns:cac="..." xmlns:cac="...">
  <cbc:ID>20080212-AA</cbc:ID>
  <cbc:IssueDate>2008-02-12</cbc:IssueDate>
  <cbc:IssueTime>13:20:54-06:00</cbc:IssueTime>
  <cac:BuyerCustomerParty>
    <cac:Party>
      <cac:PartyIdentification>
        <cbc:ID>...</cbc:ID>
      </cac:PartyIdentification>
    </cac:Party>
    <cac:BuyerContact>
      <cbc:Name>John Doe</cbc:Name>
      <cbc:Telephone>+1 555 55 55</cbc:Telephone>
    </cac:BuyerContact>
  </cac:BuyerCustomerParty>
  <cac:SellerSupplierParty>
    <cac:Party>
      <cac:PartyIdentification>
        <cbc:ID>...</cbc:ID>
      </cac:PartyIdentification>
    </cac:Party>
```

```
</cac:SellerSupplierParty>
...
</Order>
```

Example 21.3

A code snippet from a much more verbose Alleywood purchase order document.

NOTE

To view the corresponding schema used by Alleywood, see <http://docs.oasis-open.org/ubl/os-UBL-2.0/xsd/maindoc/UBL-Order-2.0.xsd> at the OASIS Web site.

To overcome the significant disparity in data representation between the Alleywood and Tri-Fold XML schemas, an XSLT stylesheet is developed, providing mapping logic that allows a runtime XSLT processor to transform the Alleywood UBL purchase order into an XML document that complies with the Tri-Fold purchase order schema.

The portion of the XSLT stylesheet that pertains to the transformation of customer data is shown here:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:UBL="...." xmlns:cac="..."
  xmlns:cac="..." xmlns:tri="..."
  exclude-result-prefixes="xsd ubl cac"
  version="2.0">
  <xsl:output indent="yes" />

  <!-- Create the Tri-Fold reference from the Alleywood ID URI. -->
  <xsl:function name="tri:ref" as="xsd:string">
    <xsl:param name="elem" as="element()" />
    <xsl:sequence select=
      "substring-after($elem/cbc:ID, '#')"/>
  </xsl:function>
  ...

  <!-- This represents the main entry point for
      the entire order. -->
  <xsl:template match="UBL:Order">
    <tri:order-form id="{ cbc:ID }" timestamp="{ date
      Time(cbc:IssueDate, cbc:IssueTime) }">
```

```
<xsl:apply-templates select=
  cac:BuyerCustomerParty, cac:Delivery,
  cac:OrderLine/cac:LineItem"/>
</tri:order-form>
</xsl:template>

<!-- Process the customer contact information. --&gt;
&lt;xsl:template match="cac:BuyerCustomerParty"&gt;
  &lt;xsl:variable name="ref"    select="
    tri:ref(cac:Party/cac:PartyIdentification)"/&gt;
  &lt;xsl:variable name="party" select="tri:party($ref)"/&gt;
  &lt;tri:customer ref="{ $ref }"&gt;
    &lt;tri:name&gt;
      &lt;xsl:value-of select="$party/name"/&gt;
    &lt;/tri:name&gt;
    &lt;tri:street&gt;
      &lt;xsl:value-of select="$party/street"/&gt;
    &lt;/tri:street&gt;
    &lt;tri:city zip="{ $party/zip }"
      state="{ $party/state }"&gt;
      &lt;xsl:value-of select="$party/city"/&gt;
    &lt;/tri:city&gt;
    &lt;xsl:apply-templates select="cac:BuyerContact"/&gt;
  &lt;/tri:customer&gt;
&lt;/xsl:template&gt;

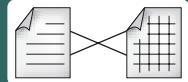
<!-- Transform the customer contact information. --&gt;
&lt;xsl:template match="cac:BuyerContact"&gt;
  &lt;tri:contact name="{ cbc:Name }"
    phone="{ cbc:Telephone }"/&gt;
&lt;/xsl:template&gt;
...
&lt;/xsl:stylesheet&gt;</pre>
```

Example 21.4

A look at the XSLT stylesheet used to enable the receipt of an Alleywood purchase order by a Tri-Fold service. Only the markup code responsible for transforming customer contact data is shown.

Data Format Transformation

By Mark Little, Thomas Rischbeck, Arnaud Simon



How can services interact with programs that communicate with different data formats?

Problem	A service may be incompatible with resources it needs to access due to data format disparity. Furthermore, a service consumer that communicates using a data format different from a target service will be incompatible and therefore unable to invoke the service.
Solution	Intermediary data format transformation logic needs to be introduced in order to dynamically translate one data format into another.
Application	This necessary transformation logic is incorporated by adding internal service logic, service agents, or a dedicated transformation service.
Impacts	The use of data format transformation logic inevitably adds development effort, design complexity, and performance overhead.
Principles	Standardized Service Contract, Service Loose Coupling
Architecture	Inventory, Composition, Service

Table 21.2

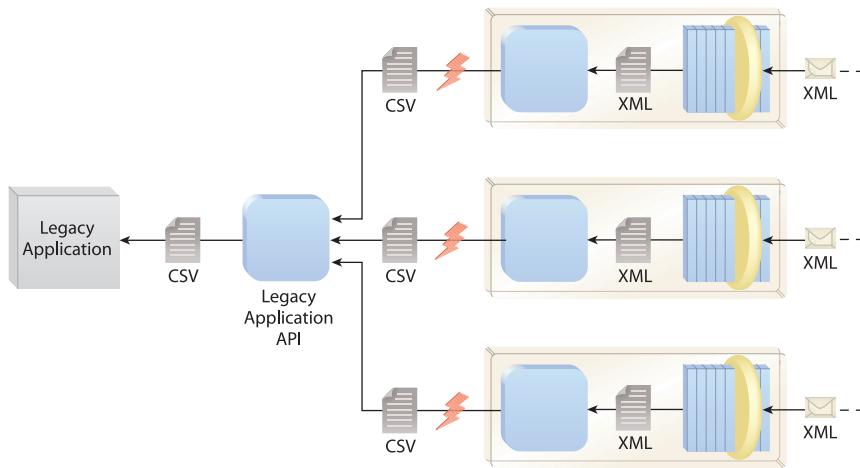
Profile summary for the Data Format Transformation pattern.

Problem

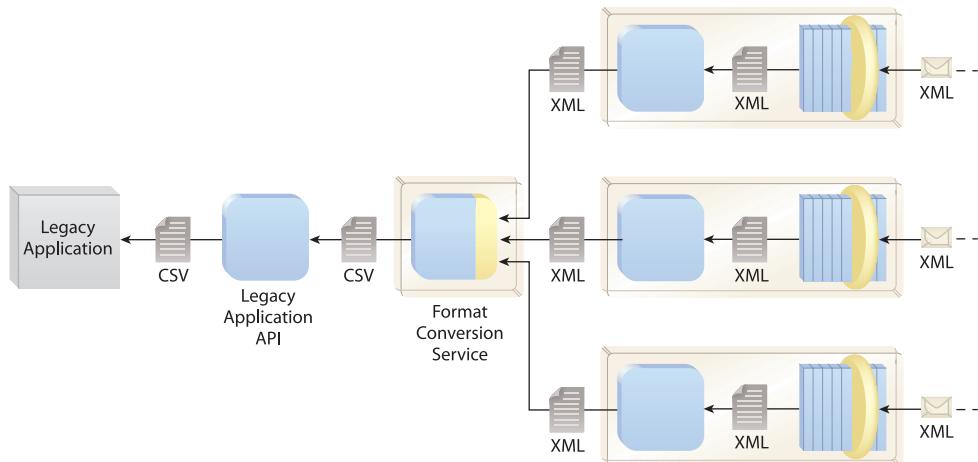
Services can be fundamentally incompatible with resources or programs that only support disparate data formats. For example, a service may have been standardized to send and receive XML-formatted data but is required to also retrieve data from a legacy environment that only supports the CSV format (Figure 21.6).

Solution

A layer of data format transformation logic is introduced. This logic is specifically designed to convert one or more data formats into one or more different data formats (Figure 21.7).

**Figure 21.6**

Three different services accustomed to processing XML formatted data need to access a legacy application API that only accepts CSV formatted data.

**Figure 21.7**

A Format Conversion utility service is added to the architecture. This service abstracts the legacy application API and provides XML-to-CSV and CSV-to-XML functions. Note that in the depicted architecture, the Format Conversion service exists as a component being reused by multiple components that are part of Web services, as per Dual Protocols (227).

Application

Data format transformation logic can exist within a service's internal logic, within a service agent, or (as shown earlier in Figure 21.7) as a separate service altogether. Middleware with brokerage features will often provide out-of-the-box services and agents that perform a variety of data format conversions.

Furthermore, because the conversion of one data format to another will naturally transform the source data model, the application of this design pattern can be seen as also applying Data Model Transformation (671). However, as previously explained, Data Model Transformation (671) is most commonly used separately for the conversion of one formal data model to another when both source and destination formats are the same.

Impacts

As with any of the three design patterns associated with the parent compound pattern Service Broker (707), the introduction of a transformation layer for data format conversion will have the following impacts:

- An increase in solution development effort when the required transformation logic needs to be custom-programmed. However, custom development effort may not be necessary when using a middleware platform already equipped with the required conversion functionality.
- An increase in design complexity, as this pattern will introduce a new layer in an already distributed environment.
- An increase in performance requirements because the format conversion will need to be executed every time interaction between disparate format sources is required. This is especially a concern with the transformation of data formats because of the tendency to carry out bulk or batch format conversions.

These impacts are often considered natural requirements that come with introducing a service layer into an enterprise with legacy applications and resources.

Relationships

Data Format Transformation is typically utilized to help integrate custom service logic with legacy systems, which makes it commonly required when applying Legacy Wrapper (441). The actual transformation logic may be the responsibility of a façade component that is tasked with converting proprietary to standardized formats, as per Service Façade (333).

When protocols are standardized using Canonical Protocol (150), data formats are usually standardized as well. This is because most communication protocols are associated with common data formats. For example the CSV format is often used with FTP and SCP protocols, and XML is primarily associated with HTTP. Therefore, when Data Format Transformation is required, it is often in conjunction with Protocol Bridging (687).

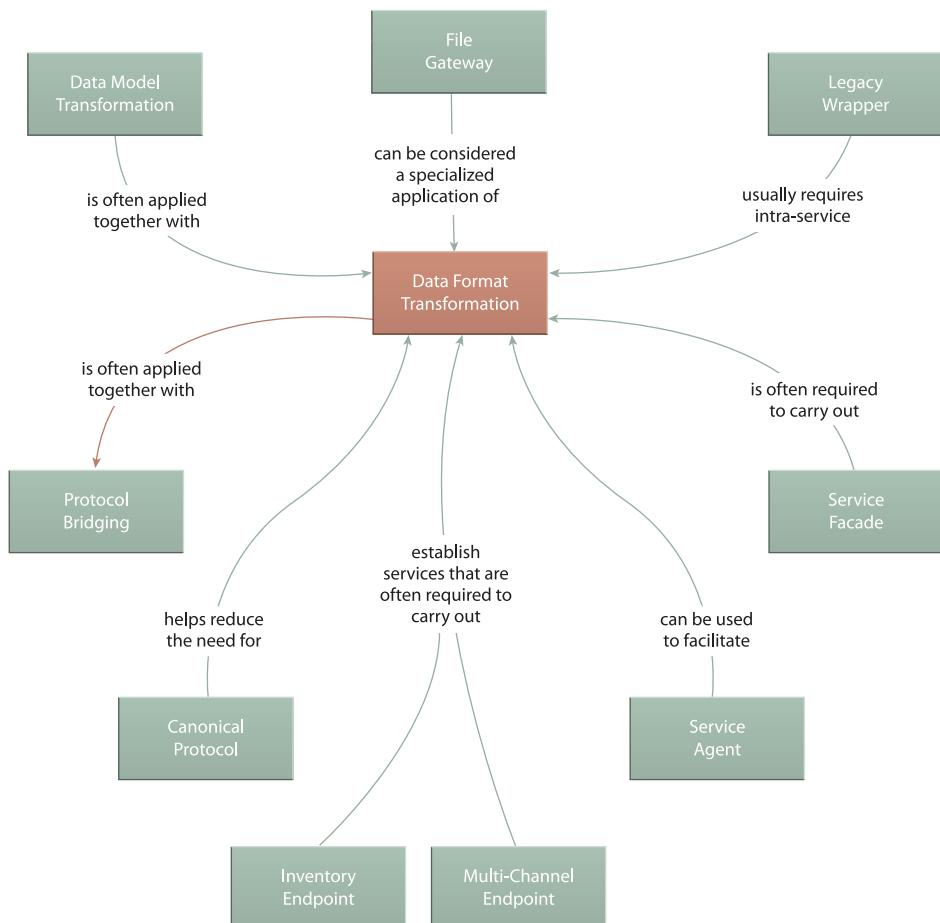


Figure 21.8

Data Format Transformation is commonly employed to deal with traditional integration issues resulting from legacy encapsulation, which is why this pattern relates to Legacy Wrapper (441) and broker-related patterns.

Also worth noting is that this pattern represents one of the three core patterns of Service Broker (707). Because Service Broker (707) is also a core part of Enterprise Service Bus (704), ESB platforms are fully expected to support the conversion of different data formats.

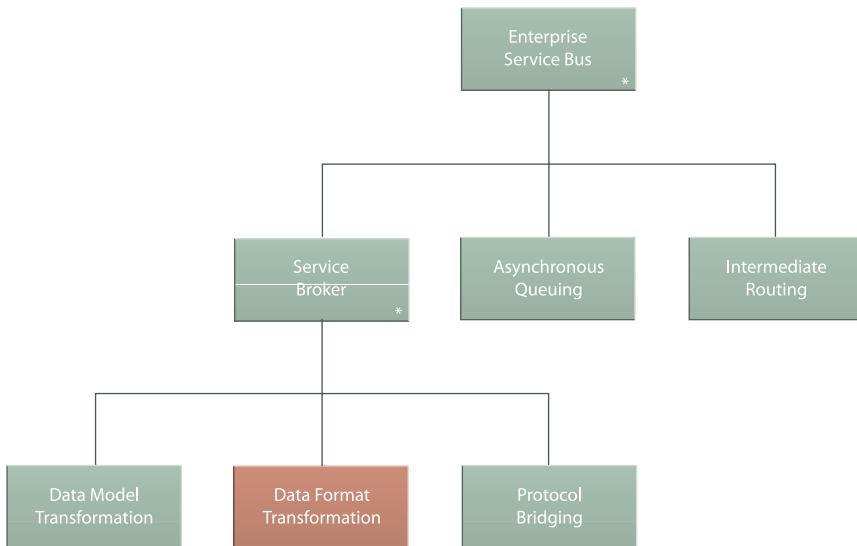


Figure 21.9

Data Format Transformation is one of three transformation-related design patterns that comprise Service Broker (707) and Enterprise Service Bus (704).

CASE STUDY EXAMPLE

Besides commercial forestry activity, another area the FRC is involved in is forestry-related disaster prevention. A large part of this program regulates critical response plans for forest wildfires.

One business process in particular governs the issuance of flight plans from fixed-wing and helicopter aircrafts capable of acting as “water bombers” for aerial firefighting. To coordinate such an effort, especially when having to call in aircrafts from different regions, a complex flight planning process is encapsulated and carried out via a central Flight Plan Validation service.

Because this service is responsible for processing flight plans sent from different sources, it has been designed to handle different data input formats. For example, the default flight plan format is XML, but the service is capable of receiving proprietary formats, such as AFTN (Aeronautical Fixed Telecommunication Network), AMHS (Aeronautical Message Handling System), as well as SMTP.

As shown in Figure 21.10, the Flight Plan Validation service converts AFTN, AMHS and SMTP message attachments to the expected XML message format, which is then forwarded to additional services.

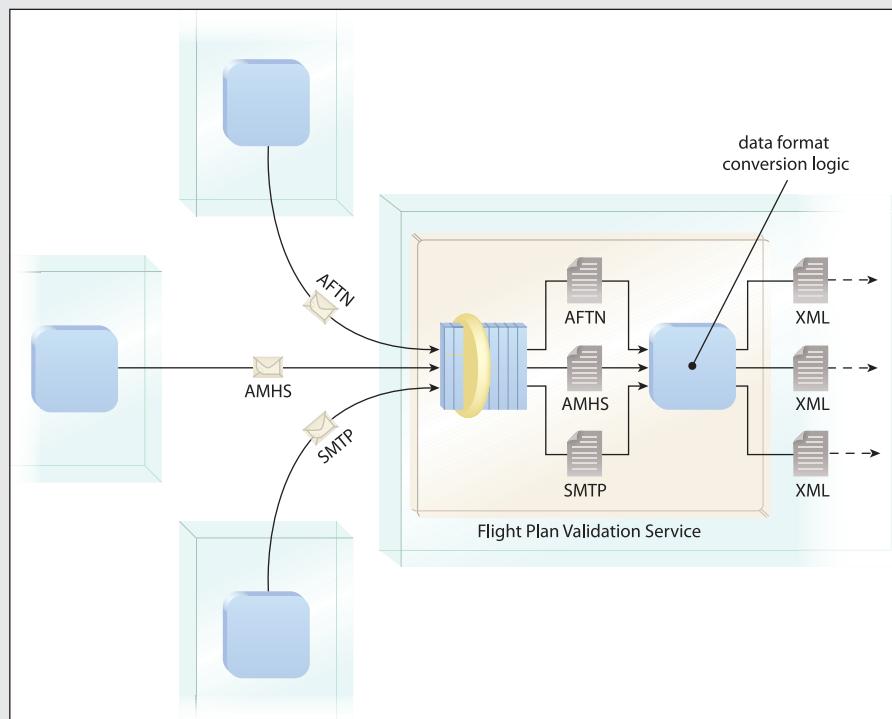
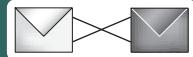


Figure 21.10

Different organizations providing critical assistance in aid of a forestry fire supply aircrafts that become part of an aerial firefighting effort. Flight plans are submitted to the FRC, which is responsible for the coordination effort. The FRC Flight Plan Validation service is designed to support the receipt of multiple flight plan format types and to then convert these formats into XML.

Protocol Bridging

By Mark Little, Thomas Rischbeck, Arnaud Simon



How can a service exchange data with consumers that use different communication protocols?

Problem	Services using different communication protocols or different versions of the same protocol cannot exchange data.
Solution	Bridging logic is introduced to enable communication between different communication protocols by dynamically converting one protocol to another at runtime.
Application	Instead of connecting directly to each other, consumer programs and services connect to a broker, which provides bridging logic that carries out the protocol conversion.
Impacts	Significant performance overhead can be imposed by bridging technologies, and their use can limit or eliminate the ability to incorporate reliability and transaction features.
Principles	Standardized Service Contract, Service Composability
Architecture	Inventory, Composition

Table 21.3

Profile summary for the Protocol Bridging pattern.

NOTE

If you haven't already, be sure to read the section *What Do We Mean by "Protocol?"* in the description for Canonical Protocol (150) before proceeding.

Problem

Services delivered by different project teams or at different times can be built using different communication technologies (Figure 21.11). For example, services can use completely disparate frameworks (such as JMS and DCOM) or different versions of the same communications technologies (such as HTTP plus SOAP versions 1.1 and 1.2).

This is a common scenario when design standards are not prevalent in an enterprise and services are primarily designed in support of tactical requirements. As a result, service interoperability outside of immediate composition boundaries is severely diminished, leading to missed opportunities to reuse and recompose services for new purposes. Over time, this

leads to “islands” of disparate service compositions that are reminiscent of traditional, silo-based application environments.

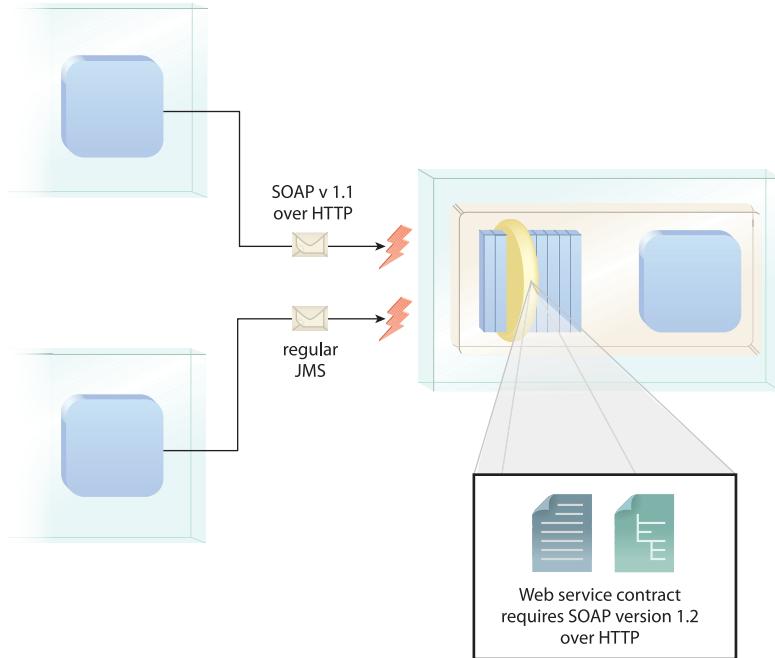


Figure 21.11

The consumer programs (left) cannot access the service because the communications protocols used by the consumers are not supported by the service. The service only accepts messages that comply to SOAP version 1.2 transmitted via HTTP.

Solution

Protocol bridging technology is used to overcome the disparity between different communications frameworks by enabling the runtime conversion of protocols (Figure 21.12).

Application

This pattern is commonly employed when legacy systems need to act as service consumers or when legacy logic needs to be encapsulated by services. As mentioned earlier, it is also used to overcome communication gaps when disparate sets of services are delivered.

A protocol bridging layer is composed of a set of adapters that act as on-/off-ramps for a given transport protocol. These adapters may be off-the-shelf products provided by the broker or a third-party vendor, or they may be generated to mirror specific service contracts using a proprietary protocol dialect.

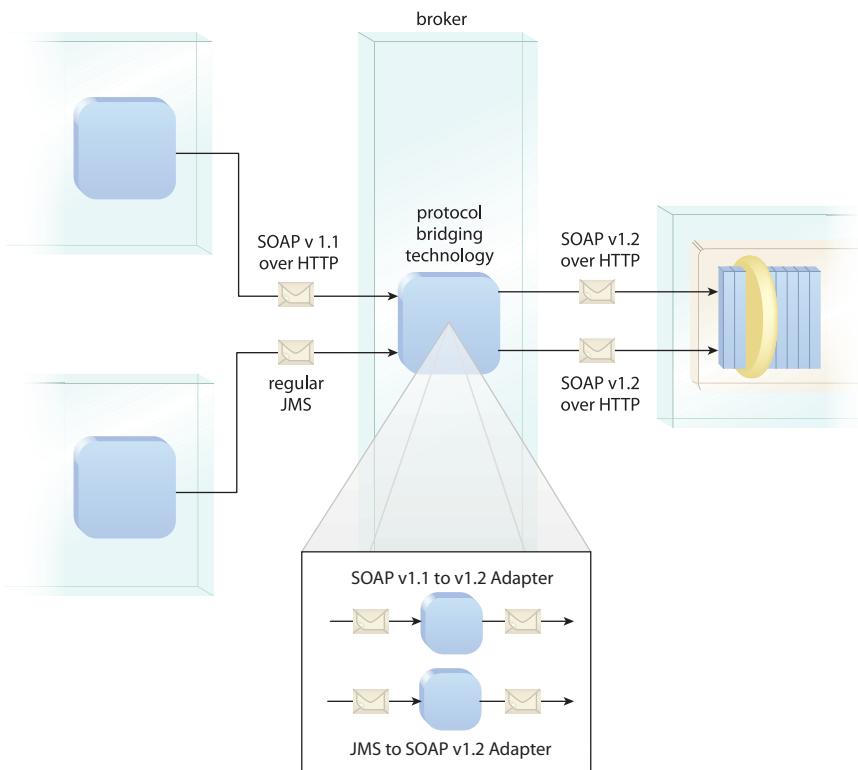


Figure 21.12

The consumer programs interact with a middle-tier broker that provides protocol bridging features. Separate protocol adapters are used to translate the two incompatible protocols to the required SOAP version 1.2 over HTTP. The broker then transmits the messages to the service on behalf of the consumers.

When a protocol bridge receives a message, it transforms it and performs whatever other work is required to make the message comply with the target protocol (or protocol version).

Protocol bridging can be used to expose or access services via multiple protocols, depending on whether adapters are implemented on the consumer or provider end. A multi-protocol conversion program can offer support for numerous protocols (HTTP, JMS, UDP, TCP, etc.) that can be natively exposed as independent protocol transformation services.

Impacts

While necessary to overcome baseline communications disparity, Protocol Bridging is a pattern only used out of necessity. From an architectural perspective, it is considered an undesirable option because of the design complexity and runtime performance overhead it imposes. Even though it can support interoperability between services, it does so at a less than optimal level. Service compositions relying on this pattern often end up having decreased levels of availability and reliability.

Protocol Bridging originated with early EAI platforms and is therefore still considered an integration-related design approach. The manner in which it involves protocol adapters and translation logic can be overly complex because of the binary encoding used by many transport protocols. In some cases, protocol conversions may even result in inaccurate semantic meanings being applied to message contents.

Though an expected part of brokerage functionality, some platforms may support limited sets of protocols with an emphasis on conversion to and from SOAP and HTTP only.

Relationships

Protocol Bridging is a pattern that has long been used to solve traditional integration problems and therefore finds itself closely related to Legacy Wrapper (441) and Dual Protocols (227). The nature of its bridging technology can also be influenced by Canonical Resources (237) when choices are available.

The extent to which this pattern is applied to intra-service bridging requirements only is based on how successful the application of Canonical Protocol (150) has been in achieving a sole inter-service communications technology.

As a part of Service Broker (707), Protocol Bridging is commonly associated with the compound pattern Enterprise Service Bus (704), as shown in Figure 21.14.

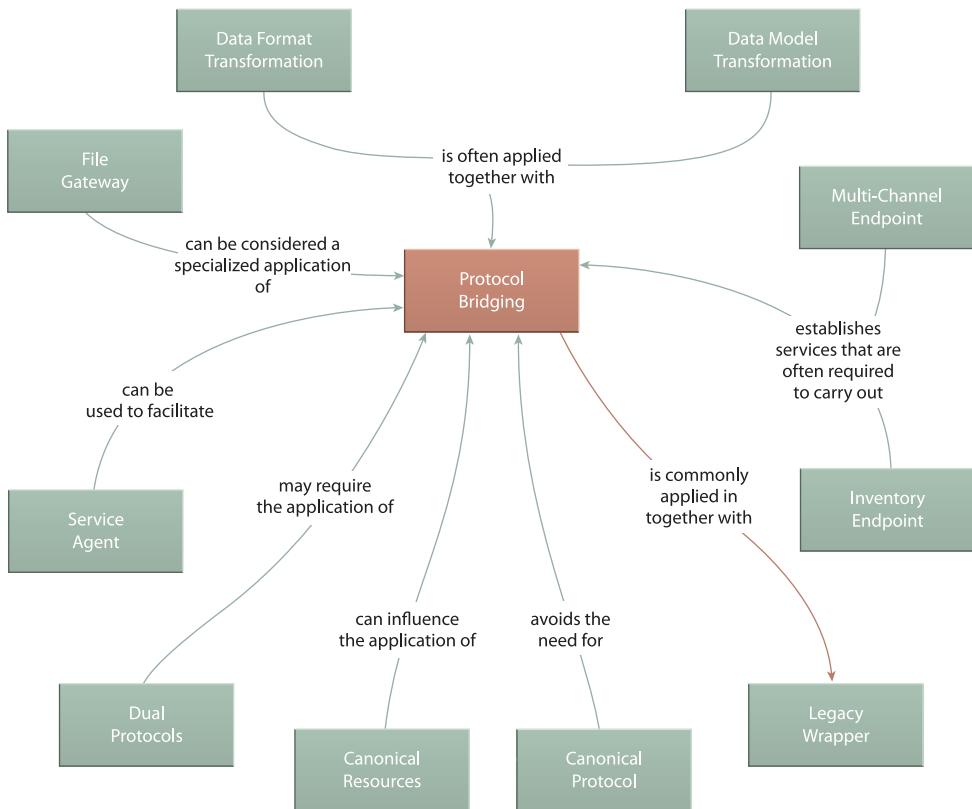


Figure 21.13

Protocol Bridging provides low-level conversion of different communication technologies and therefore relates to patterns such as Inventory Endpoint (260) and Legacy Wrapper (441) that require this type of functionality, as well as standards-based patterns that seek to avoid it.

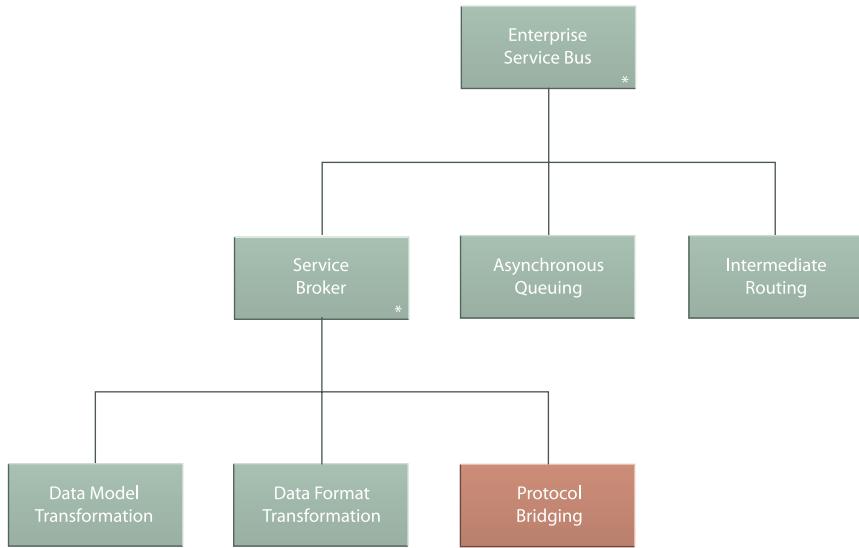


Figure 21.14

Protocol Bridging provides fundamental integration functionality that positions it as a core pattern of both Service Broker (707) and Enterprise Service Bus (704).

CASE STUDY EXAMPLE

The Cutit Run Inventory Transfer Process service composition originally depicted in the case study example for Schema Centralization (200) had been recently re-designed to incorporate standardized schemas so as to avoid the need for Data Model Transformation (671).

This composition consists of the following services:

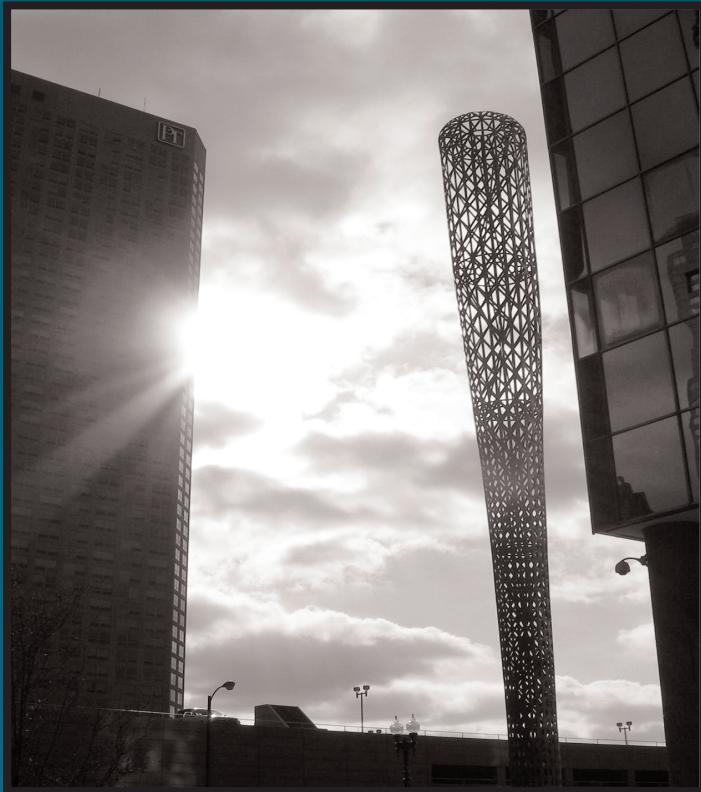
- Chain service
- Inventory service
- Order service
- Run Inventory Transfer Process service

Subsequent to receiving new requirements, the Cutit architecture team is asked to incorporate access to a legacy repository providing historical inventory data. This database can only be accessed via a vendor-provided adapter that is limited to the use of SOAP 1.1 messaging.

Cutit architects are able to apply Protocol Bridging by leveraging the broker features from their existing middleware platform to establish a means of allowing the Inventory service to query the database via its published adapter interface. However, before proceeding with this design, they decide to go a step further and apply Legacy Wrapper (441) so that the inventory archive database's proprietary API is wrapped with a standardized contract. This establishes the Inventory Archive service which provides a contract fully capable of supporting SOAP 1.2 messages by encapsulating and centralizing the necessary protocol transformation logic.

This page intentionally left blank

Part V



Supplemental

Chapter 22: Common Compound Design Patterns

Chapter 23: Strategic Architecture Considerations

**Chapter 24: Principles and Patterns at the
U.S. Department of Defense**

This page intentionally left blank

Chapter 22



Common Compound Design Patterns

Orchestration

Enterprise Service Bus

Service Broker

Canonical Schema Bus

Official Endpoint

Federated Endpoint Layer

Three-Layer Inventory

Singled out in this chapter are some of the more common and important combinations of the patterns documented in previous chapters. Each such combination is classified as a *compound design pattern*.

“Compound” vs. “Composite”

A “composite” is generally something that is comprised of interconnected parts. For example, you could legitimately refer to a service composition as a composite of services because the individual parts need to be designed into an aggregate in order to act as a whole. A “compound,” on the other hand, can simply be considered the result of combining a specific set of things together. A chemical compound consists of a combination of ingredients that result in something new when mixed together.

The patterns in this chapter are referred to as “compound patterns” because they document the effects of applying multiple patterns together. One of the most interesting parts of this exploration is that certain combinations of patterns result in design solutions that are already quite common, such as with Enterprise Service Bus (704) and Service Broker (707).

NOTE

Also worth mentioning is that a design pattern named “Composite” has been in existence for some time as a fundamental part of object-oriented design.

Compound Patterns and Pattern Relationships

Every pattern profile in the previous chapters included a *Relationships* section that provided some insight into inter-pattern dependencies and how the application of one pattern can affect another.

Compound patterns are also about relationships, but in a different way. The patterns that comprise a compound pattern have a relationship with the compound pattern itself. Whether these patterns have dependencies with or impact each other is immaterial. When studying them as members of a compound pattern, we are only interested in the results of their combined application.

Joint Application vs. Coexistent Application

When we discuss the notion of combining patterns into compounds, it is important to clarify how patterns can be combined.

A compound pattern can represent a set of patterns that are applied together to a particular program or implementation in order to establish a specific set of design characteristics. This would be referred to as *joint application*.

The compound patterns with patterns that are jointly applied are:

- Official Endpoint (711)
- Federated Endpoint Layer (713)
- Three-Layer Inventory (715)

Alternatively, the patterns that comprise a compound pattern can represent a set of related features provided by a particular program or environment. In this case, a *coexistent application* of patterns establishes a “solution environment” that may be realized by a combination of tools and technologies.

Compound patterns comprised of patterns that coexist to establish such an environment are:

- Orchestration (701)
- Enterprise Service Bus (704)
- Service Broker (707)
- Canonical Schema Bus (709)

The notation used to express compound patterns comprised of patterns that are jointly applied versus those applied in a coexistent manner differs, as shown in Figure 22.1.

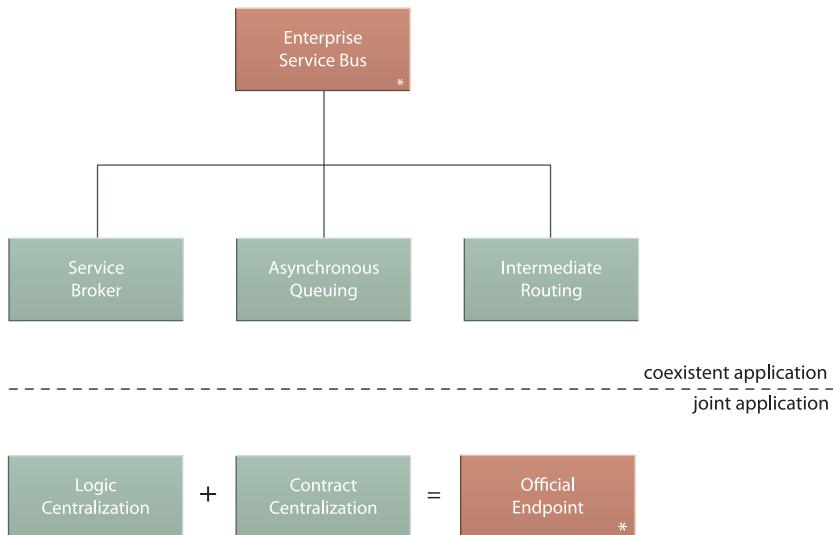


Figure 22.1

Compound patterns comprised of patterns that are applied to coexist are expressed in a hierarchy, whereas those that are made up of patterns that are applied jointly are represented via a formula-style notation.

Compound Patterns and Pattern Granularity

Even though this chapter classifies a set of patterns as “compound patterns,” it is important to note that just about any pattern can turn out to be a compound pattern. Every one of the other patterns described in this book can be decomposed into a set of more granular patterns. Their joint or coexistent combination then results in the original pattern, thereby also making it a compound pattern.

For example, Chapter 18 establishes Asynchronous Queuing (582) and Intermediate Routing (549) as providing fundamental, messaging-related solutions. In the upcoming *ESB Architecture for SOA* book, a set of more granular queuing and routing design patterns are described that can be considered as collectively representing Asynchronous Queuing (582) and Intermediate Routing (549) as compound patterns.

The reason this perspective is important is because whether or not a pattern is labeled as being a compound pattern is always relative. It is just a matter of the granularity at which the pattern is documented in relation to other patterns in the same catalog.

As a result, the compound patterns in this chapter are only classified as such because of the granularity at which the rest of the patterns are defined in this book. In a different publication or context, these patterns may be classified differently.

Orchestration

By Thomas Erl, Brian Loesgen

An orchestration platform is dedicated to the effective maintenance and execution of parent business process logic. Modern-day orchestration environments are especially expected to support sophisticated and complex service composition logic that can result in long-running runtime activities.

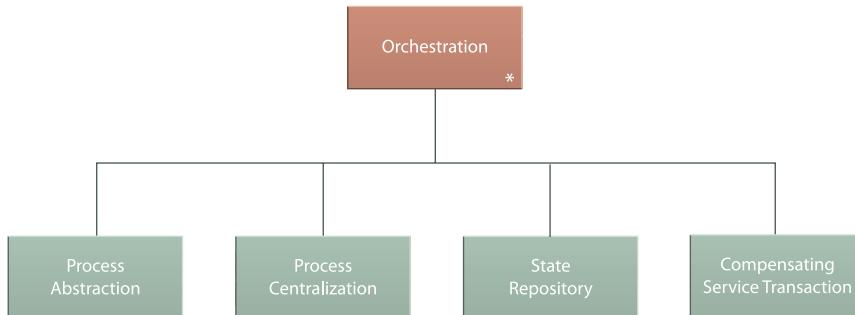


Figure 22.2

The base patterns that comprise Orchestration.

The Orchestration compound pattern is comprised of the following design patterns:

- Process Abstraction (182) is responsible for separating non-agnostic logic from agnostic logic, which forms the basis of the parent composition logic that resides within the orchestration platform and is executed by the orchestration engine.
- Process Centralization (193) limits the physical distribution of abstracted process logic into one (or a group) of locations. This allows for the centralized maintenance of parent composition logic via specialized tools provided by the orchestration platform.
- State Repository (242) enables orchestration environments to support long-running service activities by providing a native state management repository that can be leveraged as a state deferral mechanism.
- Compensating Service Transaction (631) further supports long-running processes by allowing parent composition logic to be supplemented with compensation subprocesses that address exception conditions.

Figure 22.3 illustrates how the combination of the first three of these patterns establishes an orchestration environment from an implementation perspective, and Figure 22.4 highlights how this base platform is commonly extended with features that correspond to design solutions provided by other patterns covered in this book.

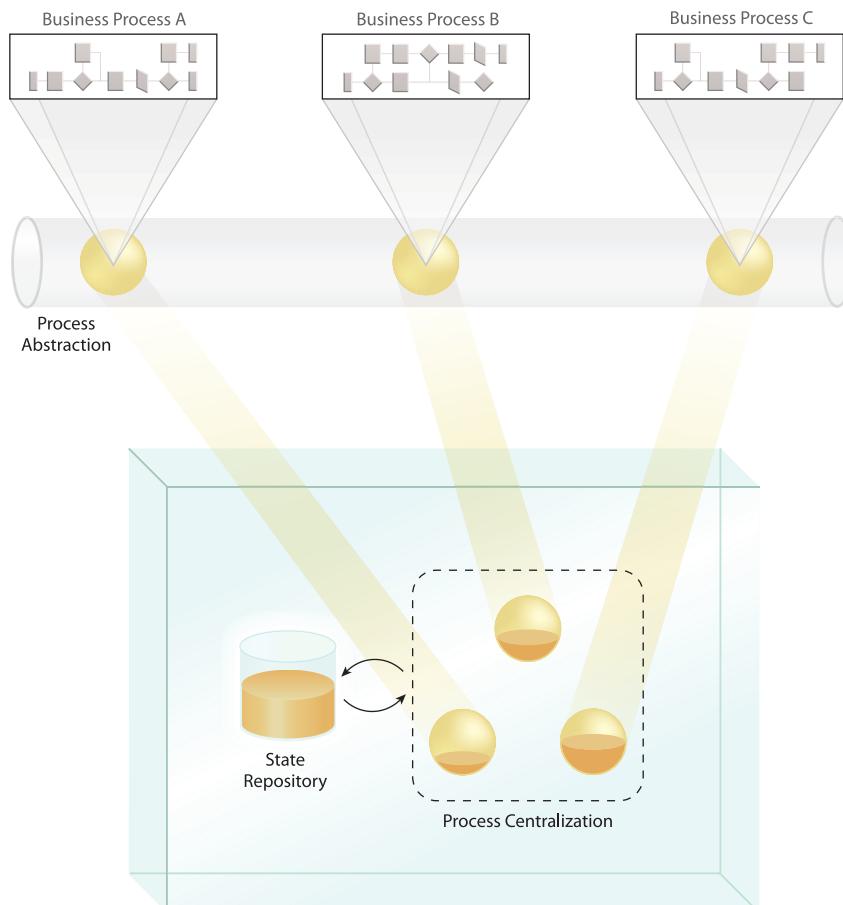


Figure 22.3

Non-agnostic business process logic is abstracted and then physically centralized into an environment that provides a native state management repository. The logic is executed and managed via an orchestration engine, which may support industry standards, such as WS-BPEL.

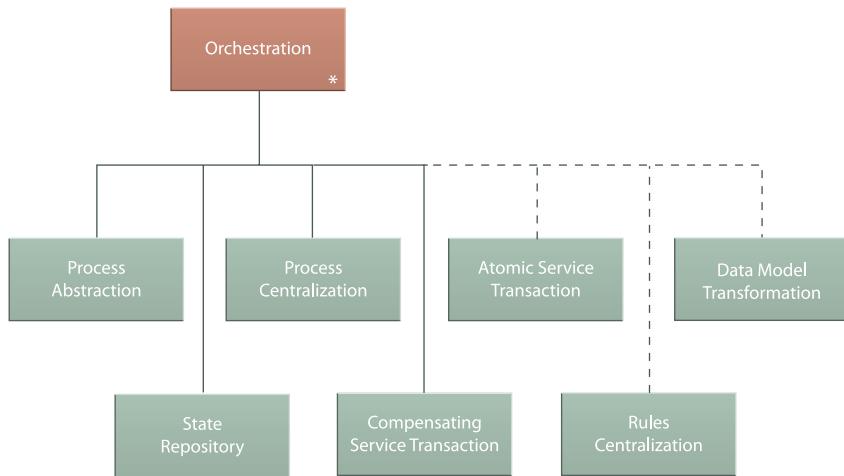


Figure 22.4

The extended Orchestration compound pattern shows how solutions provided by other design patterns expand upon the core model.

Enterprise Service Bus

By Thomas Erl, Mark Little, Thomas Rischbeck, Arnaud Simon

An enterprise service bus represents an environment designed to foster sophisticated inter-connectivity between services. It establishes an intermediate layer of processing that can help overcome common problems associated with reliability, scalability, and communications disparity.

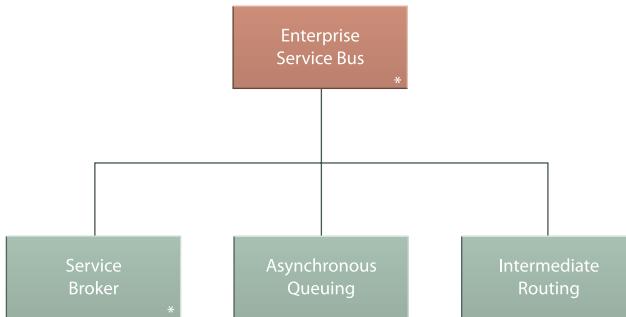


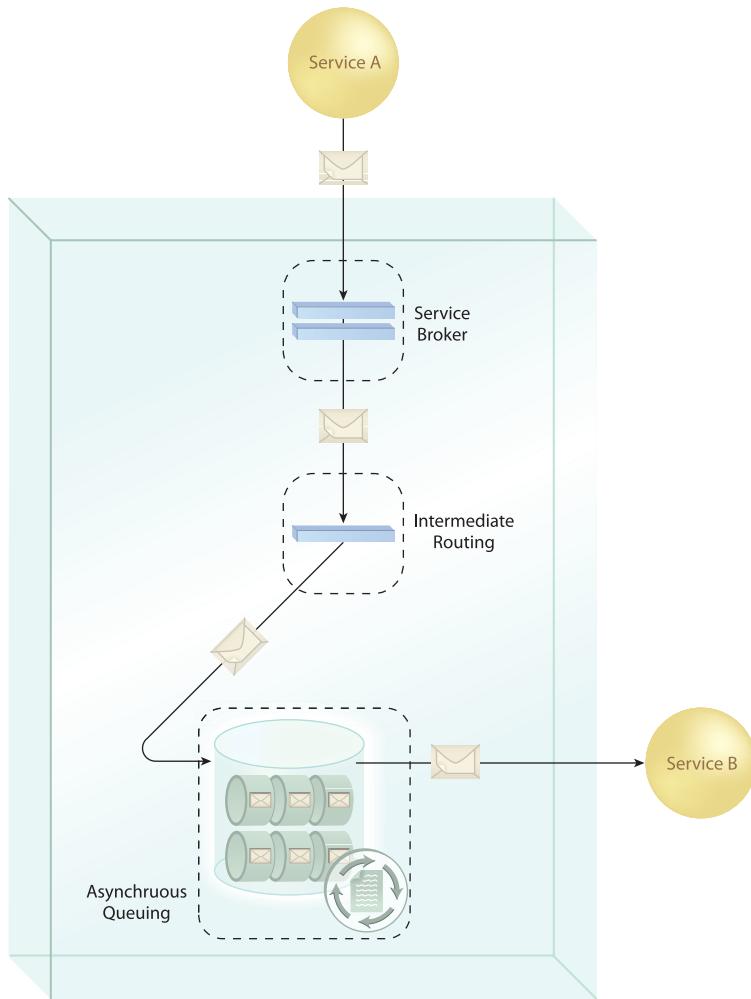
Figure 22.5

The base patterns that comprise Enterprise Service Bus.

Enterprise Service Bus is considered a compound pattern comprised of the following core patterns that coexist to provide a base set of interoperability-enablement features:

- Service Broker (707), which itself is a compound pattern that consists of a set of integration-centric patterns used to translate between incompatible data models, data formats, and communication protocols.
- Asynchronous Queuing (582), which establishes an intermediate queuing mechanism that enables asynchronous message exchanges and increases the reliability of message transmissions when service availability is uncertain.
- Intermediate Routing (549), which provides intelligent agent-based routing options to facilitate various runtime conditions.

How this set of patterns can be combined to establish a middleware-based messaging mechanism is illustrated in Figure 22.6. The expanded pattern hierarchy shown in Figure 22.7 reveals additional patterns that are common, optional extensions of Enterprise Service Bus.

**Figure 22.6**

A message undergoing various processing steps carried out by an ESB in order to successfully transform and deliver the message contents to their destination. Note that this scenario depicts all three solutions working in tandem to process a single message. Other message transmissions may not require all of this functionality. It is simply the coexistence of specific patterns that constitutes an ESB.

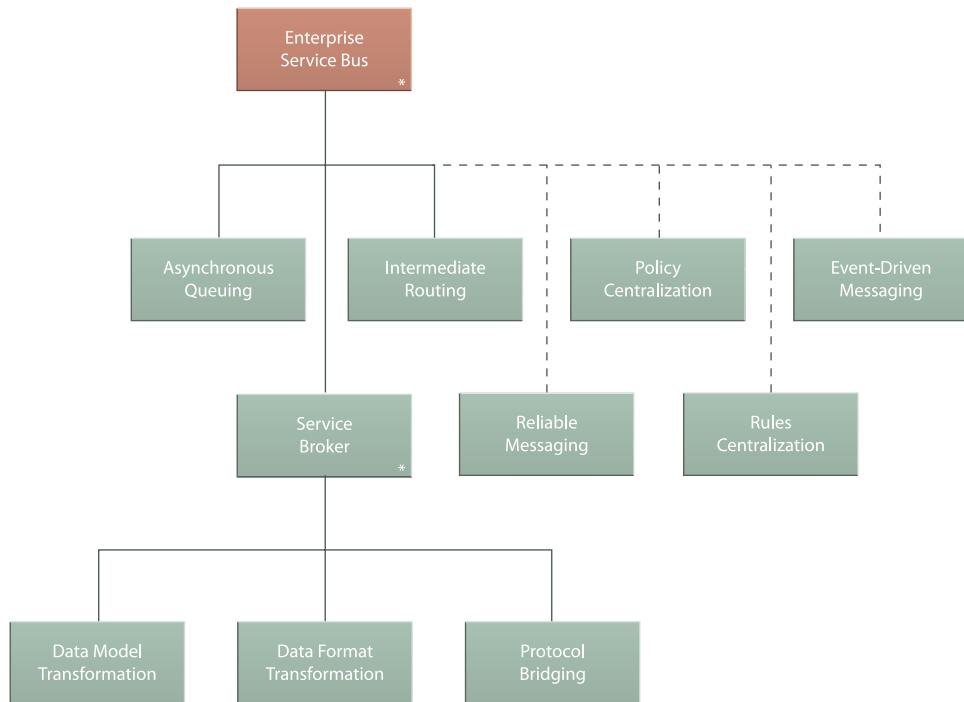


Figure 22.7

The expanded Enterprise Service Bus pattern hierarchy showing the core patterns (top left) and patterns that commonly extend the base ESB (top right), plus the patterns that comprise Service Broker (707) (bottom).

Service Broker

By Mark Little, Thomas Rischbeck, Arnaud Simon

Broker functionality has been a common part of middleware platforms, providing multi-faceted runtime conversion features that enable integration between disparate systems.

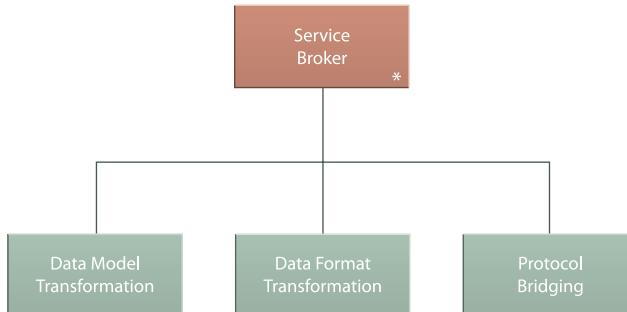


Figure 22.8

The patterns that comprise Service Broker.

The Service Broker compound pattern is comprised of Data Model Transformation (671), Data Format Transformation (681), and Protocol Bridging (687). Although all of these patterns are used only out of necessity, establishing an environment capable of handling the three most common transformation requirements can add a great deal of flexibility to a service-oriented architecture implementation, and also has the added bonus of being able to perform more than one transformation function at the same time.

Broker-related features are a fundamental part of ESB platforms, which is why this pattern exists as part of Enterprise Service Bus (704), as depicted in Figure 22.9.

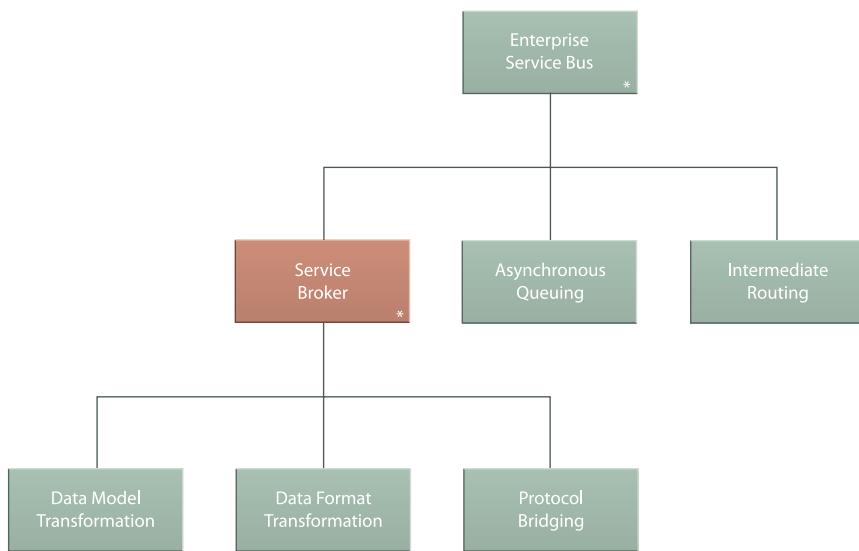


Figure 22.9

The Service Broker pattern hierarchy itself is nested within the hierarchy of Enterprise Service Bus (704).

Canonical Schema Bus

By Clemens Utschig-Utschig, Berthold Maier, Bernd Trops, Hajo Normann, Torsten Winterberg,
Thomas Erl

While Enterprise Service Bus (704) provides a range of messaging-centric functions that help establish connectivity between different services and between services and resources they are required to encapsulate, it does not inherently enforce or advocate standardization. In fact, it can be argued that the native broker functions provided by some ESB platforms can discourage standardization by making it “too convenient” to integrate disparate services.

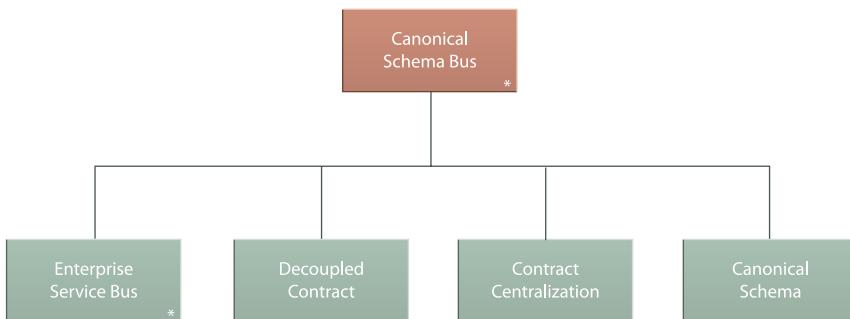


Figure 22.10

The patterns that comprise Canonical Schema Bus.

Building upon the platform established by Enterprise Service Bus (704), this pattern adds Decoupled Contract (401) and Contract Centralization (409) together with Canonical Schema (158) to position entry points into the logic, data, and functions offered via the service bus environment as independently standardized service contracts (Figure 22.11).

Canonical Schema Bus restricts entry to points to centralized and canonical service contracts and further limits the use of Service Broker (707) related patterns to intra-service transformation requirements. This places the onus of having to conform to standardized contracts upon any service or program that needs to consume them. The ultimate goal of this pattern is to promote and enforce contract-level standardization throughout a service inventory.

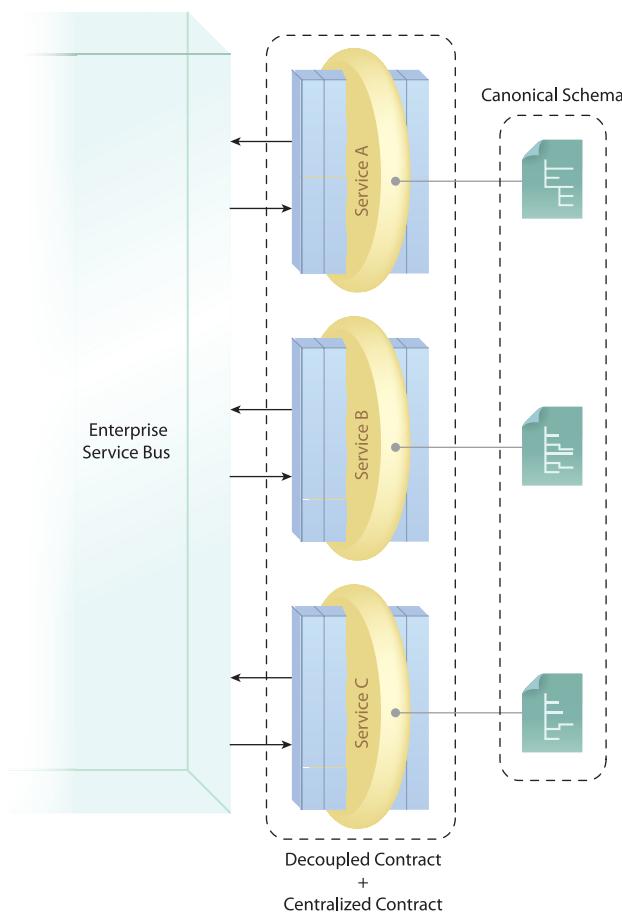


Figure 22.11

This pattern requires that access to ESB encapsulated functions and resources be limited to standardized service contracts.

Official Endpoint

As important as it is to clearly differentiate Logic Centralization (136) from Contract Centralization (409), it is equally important to understand how these two fundamental patterns can and should be used together:

- While Logic Centralization (136) asks designers to build consumer programs that only invoke designated services when specific types of information processing are required, it does not address how this logic is to be accessed.
- While Contract Centralization (409) asks designers to build consumer programs that access a service only via its published contract, it does not indicate which services should be accessed for specific purposes.

Applying these two patterns to the same service realizes the Official Endpoint compound pattern (Figure 22.13). The repeated application of Official Endpoint supports the goal of establishing a federated layer of service endpoints, which is why this compound pattern is also a part of Federated Endpoint Layer (713).

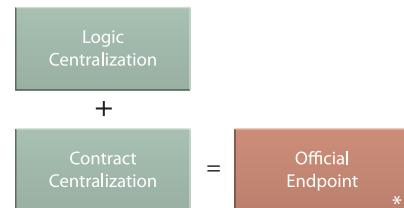


Figure 22.12

The patterns that comprise Official Endpoint.

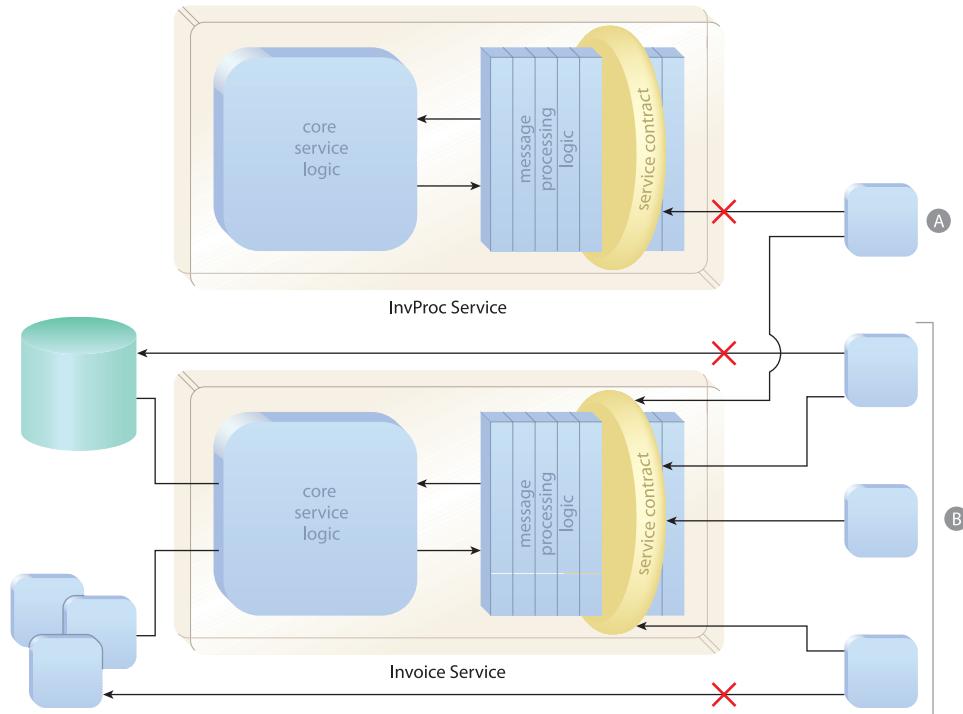


Figure 22.13

As per Logic Centralization (136), a consumer program (A) requiring invoice functionality can only access the Invoice service, which is the designated service for this logic. As per Contract Centralization (409), consumer programs (B) can only access the Invoice service via its centralized contract. The combination of these patterns establishes the Invoice service as an “official” endpoint.

Federated Endpoint Layer

Federation, within the context of this compound pattern (and also as it relates to the Increased Federation strategic goal explained in Chapter 4), represents a state where different services combine to form a united front (Figure 22.15), while allowing their respective, underlying environments to continue to be governed independently (Figure 22.16).

Federation is an important concept in service-oriented computing. It represents the desired state of the external, consumer-facing perspective of a service inventory, as expressed by the collective contracts of all the inventory's services. The more federated and unified this collection of contracts (endpoints) is, the more easily and effectively the services can be repeatedly consumed and leveraged.

The various patterns that make up this compound pattern are applied as follows:

- Each service positions the service contract as its sole entry point for a distinct functional boundary, as per Contract Centralization (409) and Logic Centralization (136) that comprise Official Endpoint (711).
- Service Normalization (131) is applied to the inventory (most likely via the prior definition of a service inventory blueprint) to ensure that no functional boundaries overlap.
- The service contracts (endpoints) themselves are standardized so that they support the same primary protocol, share the same data models for business documents, and express themselves consistently, as per Canonical Protocol (150), Canonical Schema (158), and Canonical Expression (275), respectively.

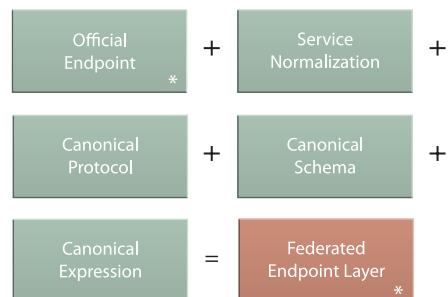


Figure 22.14

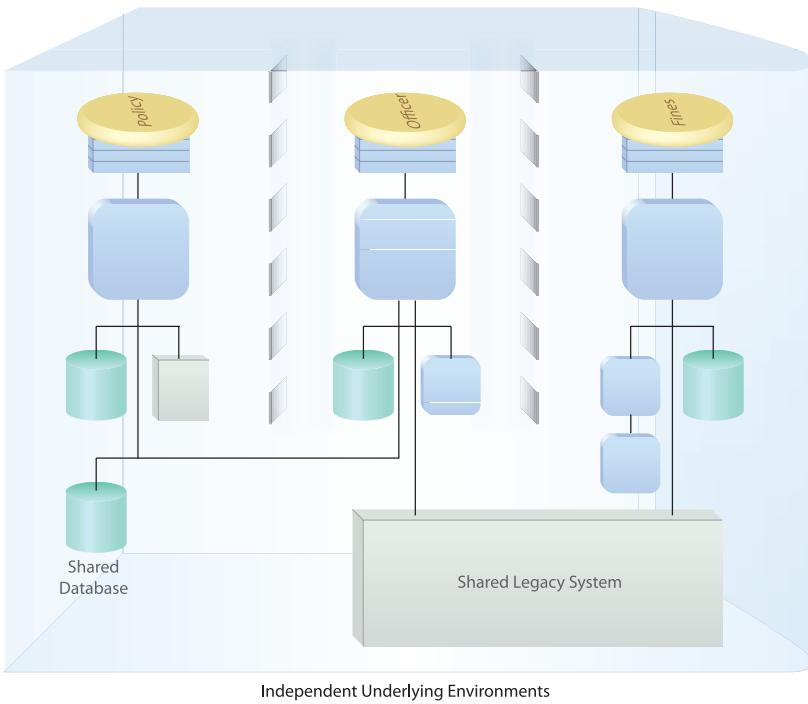
The patterns that comprise Federated Endpoint Layer.

Figure 22.15

The abstracted view of a federated service inventory, as represented by a series of standardized service contracts.



United Front

**Figure 22.16**

If you were to tip Figure 22.13 over on its side, you might see how, though providing a united front, the federated endpoints each abstract different service implementations.

Three-Layer Inventory

This compound pattern is simply comprised of the combined application of the following three fundamental inventory design patterns associated with Service Layers (143).

- Utility Abstraction (168)
- Entity Abstraction (175)
- Process Abstraction (182)

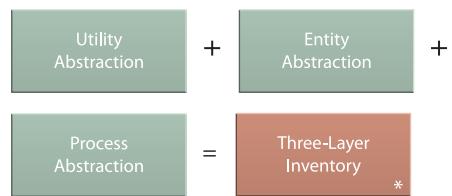


Figure 22.17

Each service layer represents a type of service based on one of three common service models.

The Three-Layer Inventory exists because the combined application of these three patterns is recommended for the following reasons:

- Each of the three abstraction patterns is proven to represent a common classification of service logic. The combined logic required to automate business processes for most organizations can be represented by these three service layers.
- The service layers are fully complementary in that each layer is distinct and does not overlap with another. As explained in the introductory section of Chapter 7, the three layers establish an effective separation of non-agnostic and agnostic plus business-centric and non-business-centric logic.
- The service models upon which these service layers are based are generic. This not only makes them common, but also customizable. For example, a typical variation of the utility service model is a state management service, as per in Stateful Services (248).

Even if an organization has unique or unconventional business requirements, it is advisable to begin a service inventory blueprint definition effort with this pattern. Once the service modeling process is underway, it will become evident as to whether defined services fit well into the three service models that establish these layers.

Figure 22.18 shows the common view of a service inventory partitioned via these three layers.

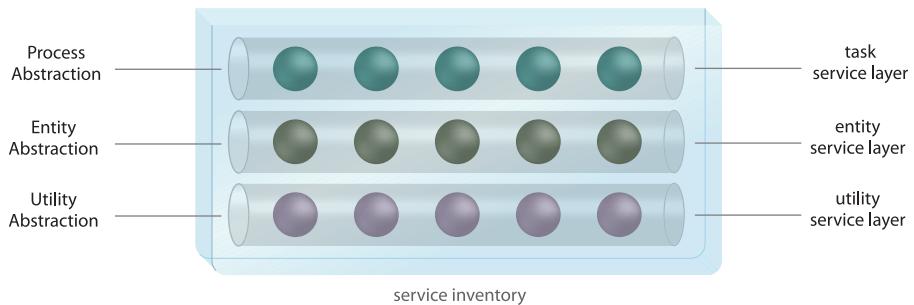


Figure 22.18

Each service layer represents a type of service based on one of three common service models.

Chapter 23



Strategic Architecture Considerations

Increased Federation

Increased Intrinsic Interoperability

Increased Vendor Diversification Options

Increased Business and Technology Alignment

Increased ROI

Increased Organizational Agility

Reduced IT Burden

All of the design patterns in this book provide design solutions in support of service-orientation. As first explained in Chapter 4, the end-result of successfully realizing service-orientation is a target state defined by a specific set of strategic goals. This chapter revisits these goals and highlights some of the key patterns that relate to their attainment.

NOTE

The patterns mentioned in each of the upcoming sections are not the only patterns that are recommended or required to achieve a given strategic goal. These sections simply provide examples in order to demonstrate how SOA design patterns can be mapped to the strategic goals of service-oriented computing. The unique requirements, environments, and constraints faced by individual IT enterprises will almost always warrant the use of additional patterns and practices. Furthermore, it's important to understand that these goals are inter-related. The attainment of one supports the attainment of others.

Increased Federation

As previously explained, federation can be classified as the unification of disparate environments while continuing to allow them to be independently governed. Within the context of service-orientation, this results in an emphasis on establishing endpoints as standardized, official “points of contact” for services (Figure 23.1).

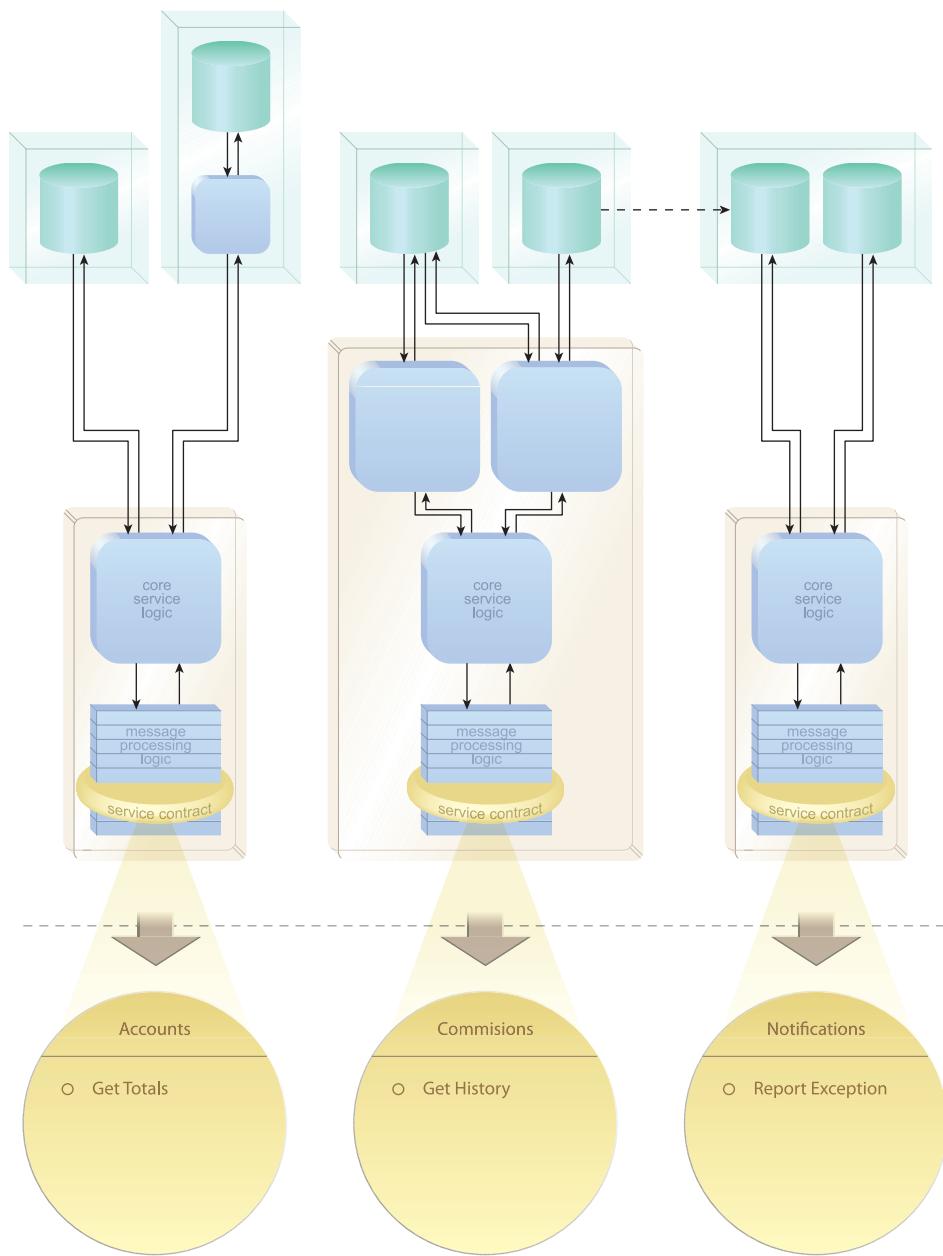
Service-orientation is focused on service contract content and the positioning of the service contract as an architectural element. This eventually results in a federated environment that needs to be attainable and maintainable by the underlying architecture. As a result, every type of service-oriented technology architecture tends to be contract-centric.

Examples of related patterns:

- Both Enterprise Inventory (116) and Domain Inventory (123) result in concrete architectural boundaries within which contract-related design standards are applied. Therefore, these two patterns establish the scope in which federation is typically pursued.

- Logic Centralization (136), Contract Centralization (409), and, as a result, Official Endpoint (711) directly support this goal by producing clear design standards that ensure that each service that is part of a federated inventory has a well-defined functional scope and can only be accessed via its standardized contract. This is further aided by Service Normalization (131), which ensures that service boundaries do not overlap.
- It is important to single out Decoupled Contract (401) as a contributor as well, as it is the physical separation of contract from implementation which helps preserve federation over time (in relation to the considerations raised by the Service Abstraction design principle).
- Service Layers (143) and its three variations tie into federation by assisting with the organization and classification of services within a federated service inventory.
- The inherent standardization that is required by contracts for a collection of services to establish a united endpoint layer is supported by patterns such as Canonical Schema (158), Canonical Protocol (150), and Canonical Expression (275). Even Canonical Versioning (286) plays a factor, especially when attempting to maintain a federated state as services continue to evolve over time.
- Of course, Federated Endpoint Layer (713) is directly based on the target state advocated by this goal, but another compound pattern that can be applied more so from an infrastructure perspective in pursuit of maintaining federation is Canonical Schema Bus (709).

In many enterprises, the extent of attainable federation can be maximized by applying these patterns to services built as Web services, primarily due to the native realization of Decoupled Contract (401) that Web service implementations naturally provide.

**Figure 23.1**

Regardless of the diversity and disparity of the underlying service implementations, from an endpoint perspective (bottom), a federated service layer establishes a consistent set of access points.

Increased Intrinsic Interoperability

The goal of increasing native interoperability represents a core state where units of service-oriented solution logic (services) are inherently compatible and therefore able to exchange data without the need to be separately integrated (Figure 23.2).

This intrinsic level of inter-connectivity relies on the federated contract-related requirements already described and can place further demands on the underlying technology architecture.

Examples of related patterns:

- On the most fundamental level, Capability Recomposition (526) establishes the baseline expectation that different services can incorporate into different compositions, thereby requiring the target state envisioned by this goal.
- From a practical perspective, contract design standards-centric patterns, such as Canonical Schema (158), Canonical Protocol (150), and Schema Centralization (200) each directly enable and enhance interoperability. Canonical Schema Bus (709) further assists this goal by enforcing contract design standards in ESB-centric environments.
- The preceding patterns and others that relate to the exchange of information between services benefit from the successful application of patterns that help position the technical service contract as an independent, yet standardized part of the service architecture. Decoupled Contract (401) and Contract Centralization (409) relate to this, as do more specialized patterns like Contract Denormalization (414) which can enhance interoperability by catering to different types of service consumers.
- Of course, Enterprise Inventory (116) and Domain Inventory (123) establish the extent to which services are expected to be natively interoperable, but Cross-Domain Utility Layer (267) is also worth mentioning as a means of improving baseline interoperability across domain boundaries.

These are just examples of patterns that help foster baseline interoperability. Because this goal is so foundational to service-orientation, in some way or another, the majority of patterns in this book support interoperability between services.

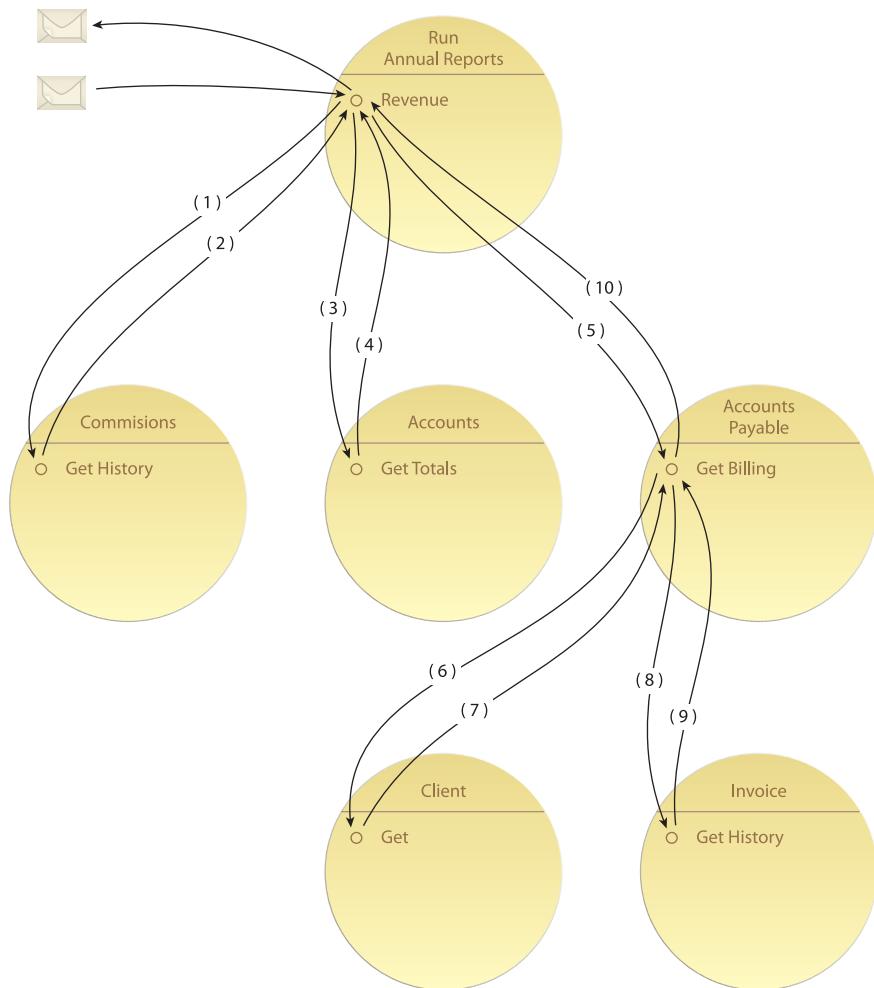


Figure 23.2

Message data travels throughout services to enable the automation of a composition. The smoother this communication, the more efficient the composition logic executes. Intrinsic interoperability strives to establish an environment in which data can be passed without constraint among numerous variations of aggregated services.

Increased Vendor Diversification Options

It is not a goal of service-orientation to increase the vendor diversity within enterprises. Instead, the objective is to provide a constant *option* for diversification so that when existing products and technologies are no longer adequate, they can be extended or even replaced with whatever else the marketplace has to offer without disrupting the established, federated service layer (Figure 23.3).

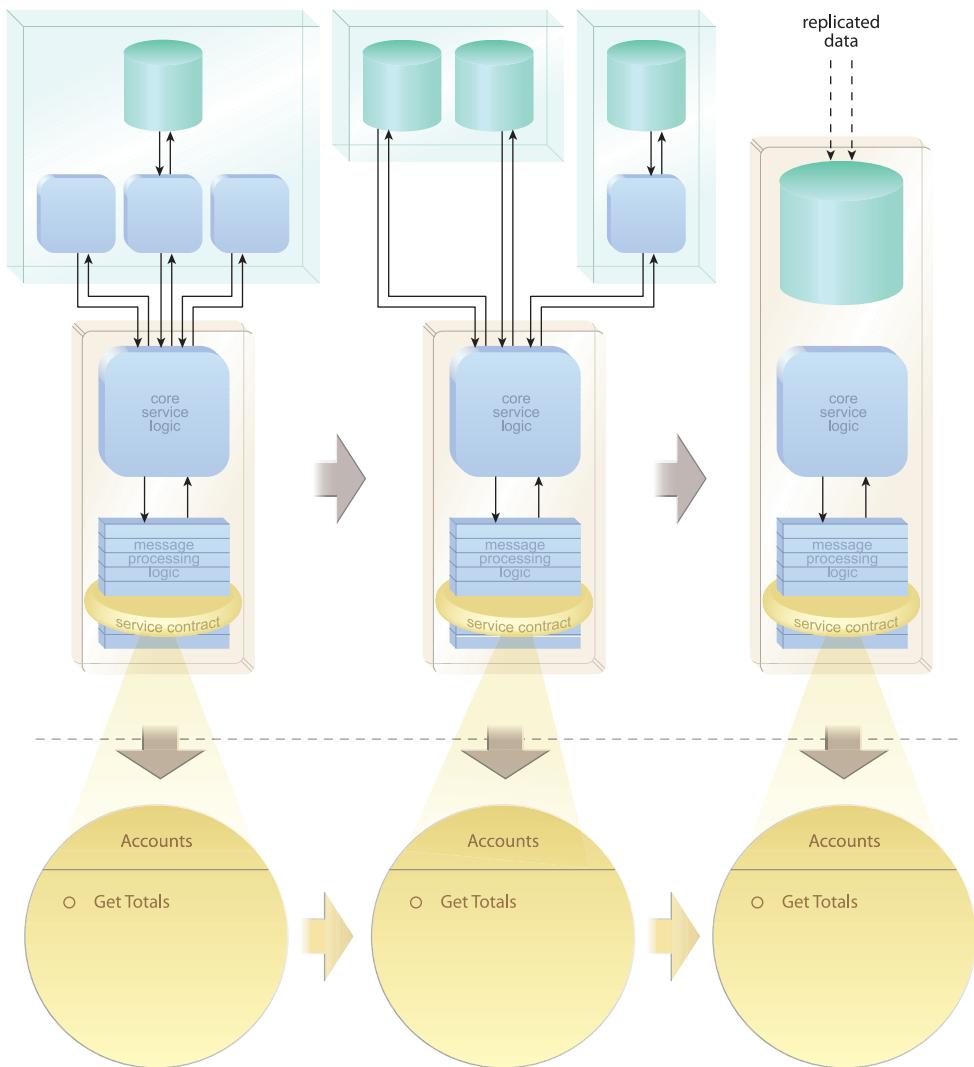


Figure 23.3

The vendor technology that underlies a service's implementation architecture may change and evolve over time, but its service contract remains the same.

The key is to allow this option to exist without having to replace a technology architecture when vendor diversification is required. This essentially imposes the requirement that architecture models remain as vendor-neutral as possible.

Examples of related patterns:

- A key design-time factor in support of this goal is how a service architecture is structured internally. For example, Service Façade (333) can help establish a healthy decoupling of various, vendor-specific resources from the core service logic so as to allow those resources to be more easily changed and replaced, as per Service Refactoring (484). On the other hand, Service Agent (543) can lead to less desirable dependencies on vendor platforms when over-applied.
- Sometimes the best solution is to wrap proprietary technology resources into non-business-centric services so as to maintain an extent of decoupling of business logic and vendor technology. For this purpose, Utility Abstraction (168) and Legacy Wrapper (441) can be applied.
- How a service relates to its underlying resources is generally determined by how it interfaces with them. Legacy-related patterns, such as File Gateway (457), can be useful in addition to the intra-service application of the transformation patterns Data Model Transformation (671), Data Format Transformation (681), and Protocol Bridging (687).
- Several patterns are simply applied via the purchase and deployment of vendor products. Examples of this include State Repository (242), Metadata Centralization (280), Service Data Replication (350), and Partial State Deferral (356).
- Patterns associated with middleware platforms, such as those established by Enterprise Service Bus (704), Orchestration (701), and Service Grid (254), can inhibit the attainment of this goal due to the fact that such platforms can encompass large portions of the infrastructure that underlie a service inventory architecture, which can lead to the architecture as a whole becoming overly dependent on supporting vendor technology.
- The preceding point ties into more specialized patterns, such as Process Centralization (193), Policy Centralization (207), Intermediate Routing (549), Asynchronous Queuing (582), Reliable Messaging (592), Event-Driven Messaging (599), Atomic Service Transaction (623), Compensating Service Transaction (631), as well as security-related patterns including Data Confidentiality (641), Data Origin Authentication (649), Direct Authentication (656), and Brokered Authentication (661).

However, it is worth noting that all of these patterns can be applied with the support of industry standards which can naturally increase the freedom to diversify vendor technologies.

- One pattern that can be considered in opposition with this goal is Canonical Resources (237). It advocates keeping underlying service implementation technologies the same in order to reduce diversity. However, the objectives of this pattern are expected to yield to the need to diversify vendor technology when having to maintain business alignment or when attempting to maximize business requirements fulfillment.

Several additional patterns can also be applied via (and are even inspired by) industry technology standards.

Increased Business and Technology Alignment

The motivation behind building a technology environment that is synchronized with the current state of a business but also designed to continuously adapt to how the business changes over time is to avoid having to reach a decision point where we have to choose between living with an outdated automation environment and replacing it altogether.

Service-orientation fosters design characteristics that enable service-oriented solutions to be extended, reconfigured, or replaced using already available resources (services) without disrupting the already established layer of federated service endpoints (Figure 23.4).

Examples of related patterns:

- In their earliest stages, services can be aligned with business intelligence as a result of service modeling processes in which foundational patterns, such as Functional Decomposition (300), Service Encapsulation (305), Agnostic Context (312), Non-Agnostic Context (319), and Agnostic Capability (324) first surface to help define the functional contexts and boundaries of services.
- The definition of services in relation to business logic is further supported by the application Service Layers (143), in particular Process Abstraction (182) and Entity Abstraction (175).

- Canonical Schema (158) and Schema Centralization (200) can also be considered supporting patterns in that they allow for the definition of independent data models for business documents that are then used and shared by business-centric services.
- From a broader and more strategic perspective, however, Capability Recomposition (526) represents the key to maintaining alignment with the business in that it is through the ability to repeatedly compose services into new aggregates that changing or new business requirements can be continually fulfilled in an effective manner.

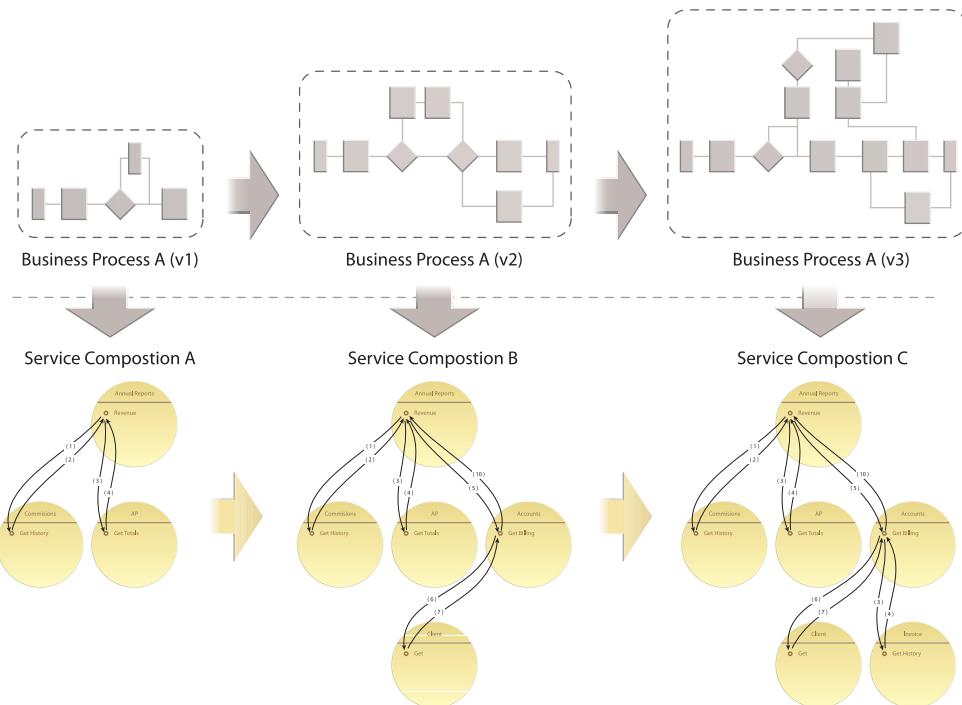


Figure 23.4

A business process (top) will evolve and change over time, and services can be continually composed and recomposed (bottom) to accommodate this change.

Increased ROI

The majority of services for a given inventory are expected to be agnostic, which means they are delivered as IT assets capable of providing repeated value to the organization over time. This leads to cost savings and a measurable return on the initial investment that was required to deliver them (Figure 23.5).

However, the extent to which each service can realize its ROI potential is directly related to its supporting service, composition, and inventory architectures.

Examples of related patterns:

- On a basic level, this goal is fundamentally supported when agnostic services and their capabilities are first defined via Agnostic Context (312) and Agnostic Capability (324).
- Agnostic services are then further formalized via the application of Entity Abstraction (175) and Utility Abstraction (168), but it's important to also acknowledge Process Abstraction (182) as means by which non-agnostic logic can be separated in support of agnostic service definition.
- Both Logic Centralization (136) and Service Normalization (131) position a body of agnostic service logic as a reusable resource while reducing the chances of overlapping logic from being introduced into the service inventory.
- Numerous patterns help establish a service architecture and supporting infrastructure that enables a given service to be reliably reused and scaled in response to increasing usage demands. Examples include Redundant Implementation (345), Service Data Replication (350), State Repository (242), Stateful Services (248), and Service Grid (254).
- Additionally, patterns like Metadata Centralization (280) and Canonical Expression (275) that help establish governance infrastructure further support the design-time identification and interpretation of services for reuse purposes.

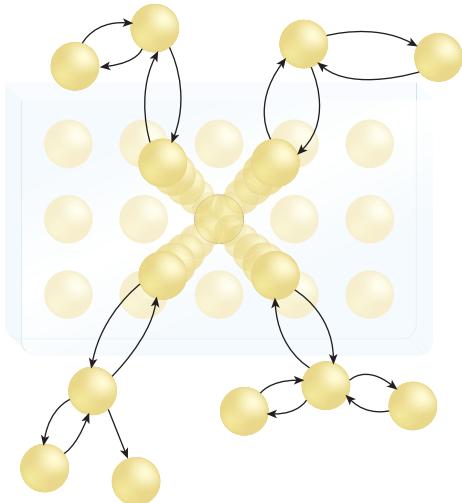


Figure 23.5

An agnostic service becomes a valuable IT asset that can be repeatedly leveraged by being reused as part of multiple compositions. In this figure, each composition invokes multiple instances of the same service.

- Finally, it is through Capability Recomposition (526) that reuse and the attainment of increased returns on initial investments are achieved.

Several other more specialized patterns can also play a key part in maximizing reuse by solving design problems specific to creating agnostic services or in relation to their runtime usage.

Increased Organizational Agility

Organizational agility refers to a state at which an enterprise is sufficiently service-oriented so that it can adapt to business change more responsively. This enables the organization as a whole to respond to new requirements or challenges with less time and effort (Figure 23.6). The result is a measurable advantage that increases its efficiency and even its overall competitiveness. To achieve this level of organizational agility, all types service-oriented technology architectures must themselves be inherently flexible and agile.

Because this strategic goal is reliant upon the attainment of several of the previously listed goals, it also benefits from the previously listed patterns that apply to those goals.

For example:

- During their inception, services modeled and designed as per Agnostic Context (312) and Agnostic Capability (324) and further shaped by variations of Service Layers (143) fundamentally establish the ability to continually apply Capability Recomposition (526) in response to new and changing business requirements
- Service Refactoring (484) and the various governance-related patterns further enable services to be more efficiently evolved and adapted.
- Runtime platform patterns, such as Orchestration (701), Enterprise Service Bus (704), and Service Grid (254), can establish effective levels of centralized operation and maintenance in addition to providing sophisticated infrastructure that allows services to be scaled and increases the overall robustness of an inventory architecture.
- The standardization realized by the successful application of canonical patterns, such as Canonical Schema (158), Canonical Protocol (150), and Canonical Versioning (286), is fully leveraged when needing to maximize organizational responsiveness with reduced solution delivery effort.

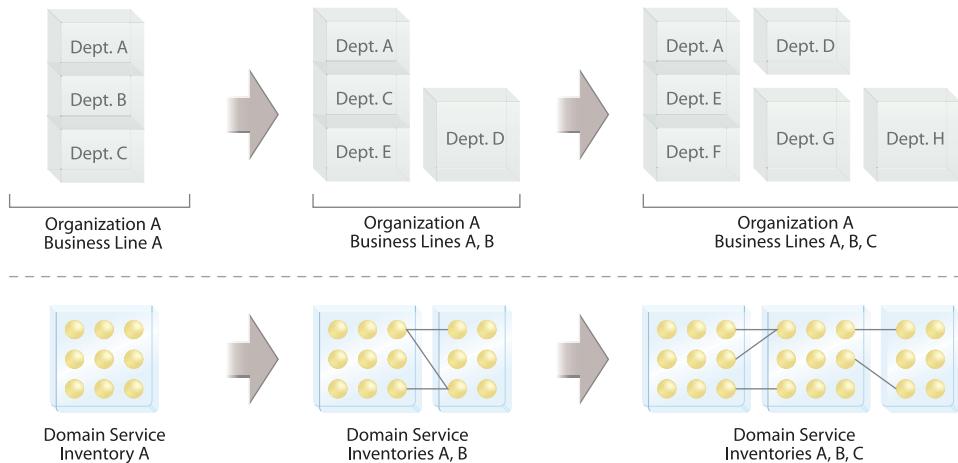


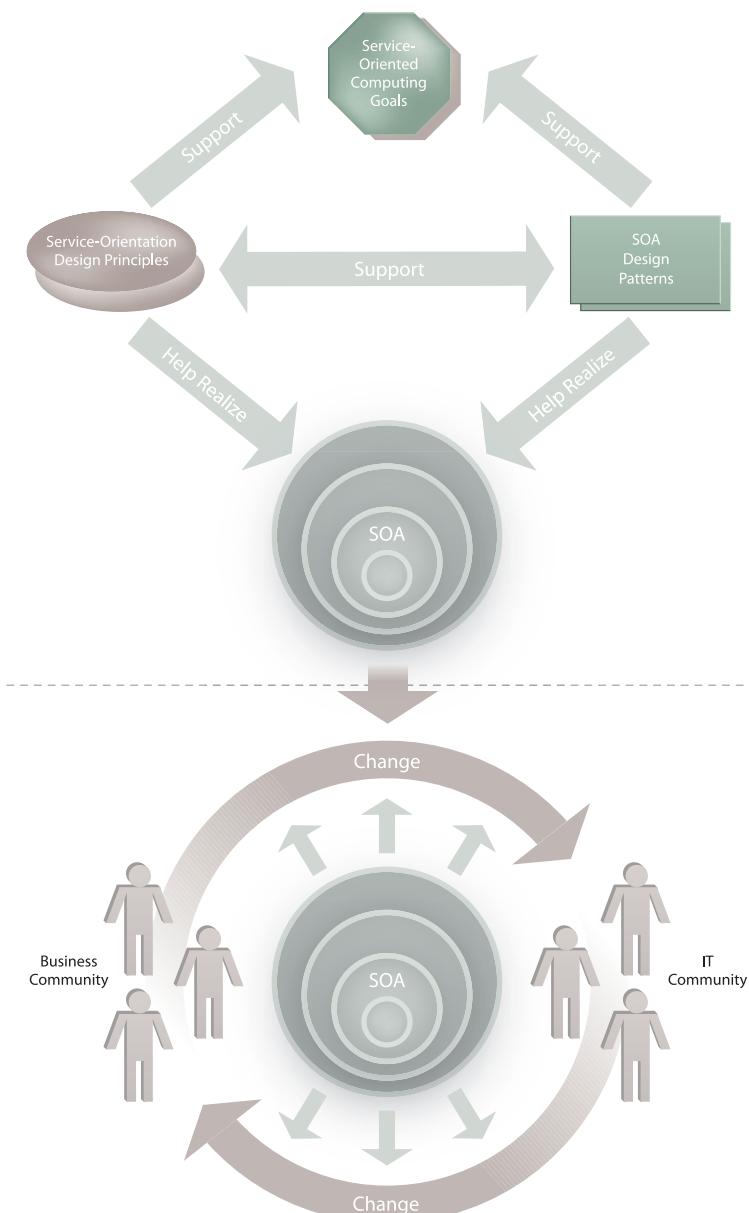
Figure 23.6

An organization will naturally evolve its business vision over time. Inventories of services can grow and can be reconfigured to support these changes, allowing the IT enterprise to move in tandem with the business.

Reduced IT Burden

This ultimate strategic benefit is the result of the combined attainment of increased ROI and increased organizational agility. A service-oriented IT enterprise can essentially offer its parent organization more responsiveness with less effort and cost.

This benefit is strategic in that for it to be continuously realized, the supporting technology environment needs to be properly governed and evolved to keep up with both business change and technology innovation. Providing the maximum business value to the organization in a manner that reduces the impact of IT is the ultimate target state of service-oriented computing. This state is achieved by realizing flexible and effective service, composition, and inventory architectures in full support of strategic business goals—and—maintaining this flexibility and effectiveness by fully leveraging every iteration of the business-technology cycle, thereby allowing both business and IT communities to evolve in support of each other (Figure 23.7).

**Figure 23.7**

The strategic goals of service-oriented computing represent a target state that service-orientation provides a method of achieving. The successful application of service-orientation helps shape and define requirements for different types of service-oriented architectures that end up establishing an IT automation model designed to fully support the endless two-way cycle of change through which business and IT communities continually transition. Amidst all of this, SOA design patterns introduce a critical success factor by providing proven design solutions and practices that support (and are supported by) service-orientation.



Chapter 24

Principles and Patterns at the U.S. Department of Defense

By Dennis Wisnosky, Chief Architect, Business Mission Area,
U.S. Department of Defense

The Business Operating Environment (BOE)

Principles, Patterns, and the BOE

The Future of SOA and the DoD

SOADoD.org

“The Department of Defense (DoD) is perhaps the largest and most complex organization in the world. It manages more than twice the budget of the world’s largest corporation, employs more people than the population of a third of the world’s countries, provides medical care for as many patients as the largest health management organization, and carries five hundred times the number of inventory items as the world’s largest commercial retail operation.”

—DoD Enterprise Transition Plan, September 2005

In 2005, the U.S. Congress passed the National Defense Authorization Act for FY2005 to modernize and streamline automated business systems within the DoD with the purpose of improving organizational agility and increasing the value of IT in general, while also reducing the historically escalating operational IT costs.

This legislation resulted in the creation of the Business Enterprise Architecture (BEA), which serves as the master blueprint for guiding and constraining the DoD’s investments in business systems. These investments total to approximately 53% of the DoD’s IT budget, roughly equivalent to the sum of the remainder of the budget for the entire U.S. Federal Government.

In 2006, the Business Mission Area (BMA) decided to plan and manage these investments via an architectural approach based upon SOA as a means of achieving modularity, interoperability, and the DoD’s overall goals of “net-centricity” and disciplined information sharing.

NOTE

Net-centricity is the realization of a networked environment that includes infrastructure, systems, processes, and people. Its purpose is to enable net-centric operations—a completely different approach to warfighting, intelligence, and business functions.

Through SOA, the DoD's business IT solutions are being united via an infrastructure and standards-based pattern termed the *Business Operating Environment (BOE)*. The other DoD mission areas (Warfighting, DoD Intelligence, and the Defense Information Environment) are following this same approach. As a result, many of the projects and inter-related sub-projects that drive these mission areas are fundamentally based on the development of services through SOA and service-orientation.

The Business Operating Environment (BOE)

Due to the scale, complexity, and diversity faced by project teams when progressing with SOA adoption and roll-out efforts, the DoD developed a strategy with guiding principles that correlate with service-orientation design principles and with approaches that apply many established SOA design patterns. This strategy is represented by the BOE, and because it encompasses many SOA design patterns, it can therefore itself be considered compound pattern (Figure 24.1).

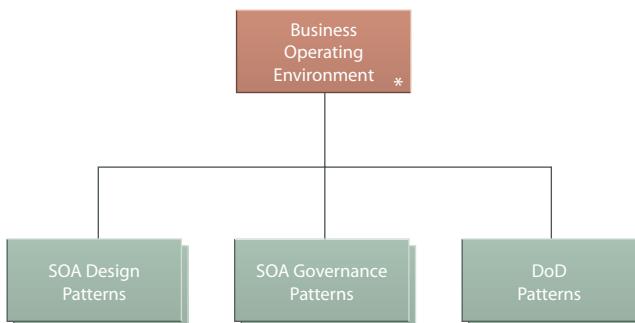


Figure 24.1

The BOE can be represented as a broad compound pattern encompassing SOA design and governance patterns, as well as a number of patterns specific to the DoD.

The emphasis of the BOE is to enable the incremental, phased adoption of SOA within DoD enterprise environments by framing individual service-oriented architecture implementations in support of the overarching business vision and supporting enterprise architectures. Within the BMA, it is intended as a key realization of the DoD's vision for a Global Information Grid and to also integrate and leverage the DoD common SOA infrastructure by establishing a set of core enterprise services and standards to be used across all DoD Components (departments, divisions, organizations that comprise the DoD) and Mission Areas.

This framing approach essentially relies on the identification of the key parts and requirements of an effective service-oriented technology architecture implementation and then guides the development, acquisition, and incorporation of services across multiple business units within the overall enterprise. As such, it sets a model for various parts of the enterprise to follow as they transition to and standardize on service-orientation. The BOE defines and helps propagate a common shared vision of how IT enterprises create and deploy services, resulting in a target state whereby shifts in business and IT strategy can be accommodated at the speed of service composition.

Within the DoD, the BOE essentially establishes a modernized, services-based IT ecosystem.

Principles, Patterns, and the BOE

The BOE is based upon a set of guiding principles formulated to keep the delivery of business capabilities aligned with the net-centric guidance of the DoD while providing support for business transformation. BOE guiding principles individually correspond and relate to a number of SOA design patterns, as well as the service-orientation principles (Figure 24.2) documented in Thomas Erl's *SOA Principles of Service Design*.

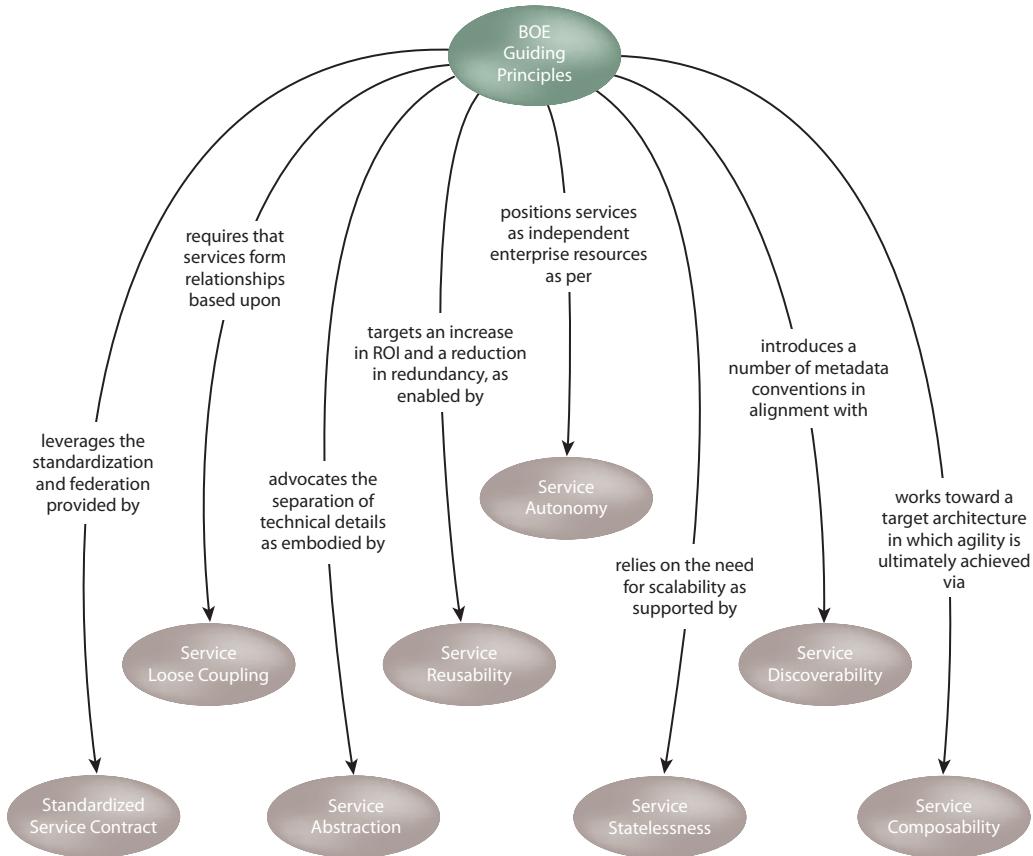


Figure 24.2

BOE guiding principles have a strong relationship with service-orientation principles and share many common goals.

The following sections highlight how BOE guiding principles relate to SOA design patterns and service-orientation design principles.

NOTE

To avoid confusion between BOE and service-orientation principles, BOE principles are referred to as “guiding principles” in the upcoming sections.

Incorporation of Information Assurance (IA)

Information assurance relates specifically to information security. This guiding principle states that the development and application of the BOE will incorporate IA requirements as a core part of the DoD infrastructure and in conformance with pre-defined security standards and directives. Security patterns, such as Direct Authentication (656), Brokered Authentication (661), Data Confidentiality (641), and Data Origin Authentication (649), can play a key role in supporting the goals of this guiding principle.

Adherence to Standards

Vendor neutrality and openness is advocated throughout a technology architecture and realized through the adherence to open standards for consistent system and data interoperability. This is supported by the Standardized Service Contract principle but can also be associated with service-orientation as a whole when it represents the standard paradigm for solution design.

Data Visibility, Accessibility, and Understandability to Support Decision Makers

In support of attaining the goals of DoD Business Transformation and carrying out various related activities (including those described in the DoD Net-Centric Data Strategy and the DoD Net-Centric Services Strategy), business data and services need to be easily located, understood, and reused by authorized users.

To follow this guiding principle, pre-defined protocols and other forms of standardization are applied to information sharing in general. While this is directly supported by the Standardized Service Contract, Service Discoverability, and Service Reusability principles, it also requires the successful application of patterns such as Canonical Schema (158), Contract Centralization (409), and Canonical Protocol (150) to ensure consistent interoperability.

When working with data from legacy repositories, common Service Broker (707) patterns, such as Data Model Transformation (681) and Protocol Bridging (687), may be further required. Also worth noting is that due to their utility-centric nature, data services are generally reliant upon the application of Utility Abstraction (168).

Loosely Coupled Services

The BOE is very much focused on the creation of independent business services in support of the overall DoD SOA initiative (and building upon core infrastructure capabilities provided by the Defense Information Systems Agency). Its ultimate goal is to establish a

“suite” of loosely coupled services and service-oriented solutions to automate cross-enterprise business processes.

While this is clearly aligned with the Service Loose Coupling design principle, it is also supported by the Service Autonomy principle due to the emphasis on establishing independent services that can be readily composed. Service Statelessness may also play a part in realizing the goals of this guiding principle when scalability issues arise.

Because the goal of establishing collections of loosely coupled services is foundational to service-oriented computing in general, this guiding principle can be supported by the application of most SOA design patterns.

Authoritative Sources of Trusted Data

Metadata repositories should be provided for business data asset and service producers so that such data sources can be registered and become visible to potential consumers. When associated with the discovery of data services this can relate to the Service Discoverability principle as well as Metadata Centralization (280).

Positioning data assets and services as authoritative sources of data relies upon the application of Official Endpoint (711), the compound pattern comprised of Logic Centralization (136) and Contract Centralization (409).

When following this guiding principle on a data architecture level, Schema Centralization (200) can become necessary to ensure that individual document schemas provide standardized representations of data sources as well as providing data service consumers with the means to understand the data provided.

Metadata-Driven Framework for Separation from Technical Details

Users and consumers are separated from technical and implementation details by requiring access to only the metadata describing the character and invocation interfaces of the services, as opposed to the underlying technology platforms and approaches used for implementation. This builds upon the search, discovery, and registry extensions established by the preceding guiding principle to create an environment wherein service implementations can be independently governed and evolved without impacting those that use and bind to them.

The Service Abstraction principle embodies the fundamental concepts of information hiding in support of these goals so that services are positioned as black boxes that can be subject to a variety of governance patterns, such as Service Refactoring (484) and Service Decomposition (489).

Support Use of Open Source Software

Open source software solutions will be viewed and used on an equal basis with regular commercial offerings, thereby enabling the DoD to leverage the cost savings and source code availability of open source software.

With the increasing amounts of services-centric open source projects, following this guiding principle opens the door to applying the Service Reusability and Service Composability principles in new ways by building pure open source or hybrid (open source plus commercial) service-oriented solutions.

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

For COTS product acquisitions to be considered, they must provide built-in support for standardized service interface contracts or for readily producing the same from COTS services that can be customized and pulled into existing service inventories, as per the policies and standards outlined as part of the BOE. A related aspect of this requirement is that COTS functions or capabilities that duplicate common functionality provided by the DoD and BMA enterprise services should be factored out in favor of invoking or using the DoD services.

This relates to several service-orientation principles, including Service Reusability, Service Composability, and Standardized Service Contract. To overcome limitations within (existing or new) COTS products, it may also be necessary to apply legacy-centric patterns, such as Legacy Wrapper (441) or any of the Service Broker (707) patterns.

Participation in the DoD Enterprise

Services produced by projects that follow the BOE approach are in compliance with the overarching DoD enterprise and can interoperate with existing services, such as the DoD enterprise services, regardless of tier.

Principles focused on repurposing services (such as Service Reusability, Service Discoverability, and Service Composability) are critical to achieving this goal, as are core design patterns associated with inter- and intra-inventory interoperability.

Support Mobility — Users & Devices

BOE technology and services should support a wide range of mobile and intermittently-connected devices. This requires that the Service Reusability principle be applied with different types of consumers in mind. Multi-Channel Endpoint (451) can directly address the requirements of this guiding principle, especially when legacy encapsulation is part of the overall service architecture.

The Future of SOA and the DoD

As SOA adoption continues to grow throughout DoD Components (the Office of the Secretary of Defense, the DoD Services, the Combatant Commands, DoD Agencies and Field Activities, etc.), so does the awareness of service-orientation and the necessity for standards, patterns, and principles to be applied appropriately and consistently. In support of these objectives, the BOE remains the driving and guiding force behind the numerous SOA projects that are currently underway in the BMA.

SOADoD.org

There are numerous public documents available about the SOA initiatives at the U.S. Department of Defense. SOADoD.org is a new portal site dedicated to providing specifications, project descriptions, and other resources pertaining to the on-going SOA adoption effort at the DoD.

—Dennis Wisnosky, Chief Architect, Business Mission Area,
U.S. Department of Defense

This page intentionally left blank

Part VI



Appendices

Appendix A: Case Study Conclusion

Appendix B: Candidate Patterns

Appendix C: Principles of Service-Orientation

Appendix D: Patterns and Principles Cross-Reference

**Appendix E: Patterns and Architecture Types
Cross-Reference**

This page intentionally left blank

Appendix A



Case Study Conclusion

The case study examples provided throughout the chapters in this book have demonstrated the application of patterns in a diverse variety of scenarios, while also providing insight into the technologies commonly used for a given pattern. The following sections briefly summarize the effects of key patterns applied to the three organizations that were studied. Each section also discusses how these patterns helped attain the business goals of the organization, as originally established in Chapter 2.

Cutit Saws Ltd.

The justification for Cutit to invest in an SOA initiative was based on a very clear goal of wanting to take advantage of the recent increase in their product demand and to parlay this into prolonged revenue growth so as to position their company as a prime acquisition target.

The application of the foundational service patterns Functional Decomposition (300), Service Encapsulation (305), Agnostic Context (312), Non-Agnostic Context (319), and Agnostic Capability (324) helped Cutit define a set of services to automate their previously archaic supply-chain process. The resulting Inventory, Chain, Order, and Run Chain Inventory Transfer services were combined into a service-oriented solution that addressed the key goal of improving the automation of their inventory transfer process, while establishing individual services that were ready-made to adapt to future business change.

These and other services were further refined via Canonical Expression (275), and optimized through the application of patterns such as Messaging Metadata (538), Service Callback (566), Service Instance Routing (574), and Compensating Service Transaction (631). The overall Cutit service inventory was also well structured and standardized with the help of Schema Centralization (200) and Policy Centralization (207).

Alleywood Lumber Company

The Alleywood IT enterprise found itself in a more complicated situation, having been acquired by McPherson and forced to find ways to cooperate and interoperate with the IT department of the also-acquired Tri-Fold company. Alleywood's strategic goals were primarily focused on ensuring that the merging and integration of its IT assets within the overall McPherson enterprise was successful.

Throughout the examples, Alleywood faced numerous situations that were solved by the application of design patterns. Perhaps the most significant was the decision by McPherson architects to proceed with Domain Inventory (123), which allowed the Alleywood enterprise to build and govern its own service inventory independently from the one established by Tri-Fold.

While this empowered Alleywood architects to build services and proceed with their SOA adoption on their own terms, overarching standards were still applied by McPherson architects via Canonical Protocol (150) and Canonical Schema (158) across both Alleywood and Tri-Fold inventory domains.

This innovative strategy of defining domain service inventories while still achieving an extent of cross-domain standardization enabled McPherson to attain a high level of baseline interoperability across all services. This, coupled with the cross-domain reuse achieved by Cross-Domain Utility Layer (267) and the master service registry established by Metadata Centralization (280), laid a solid foundation for the attainment of the long-term strategic goal of creating a global IT environment that is diverse, yet still harmonized and streamlined.

While designing and delivering various services, the Alleywood team faced many design problems that were addressed with the help of design patterns. The most memorable scenario perhaps was the security breach that they had to contend with when their Retail Lumber service was attacked and eventually decommissioned. The resulting cooperative effort between Alleywood and McPherson architects and security specialists resulted in a project during which Exception Shielding (376), Trusted Subsystem (387), and Message Screening (381) were applied to produce a vastly improved service design that was protected well beyond the attacks it had previously been subjected to.

Another significant stage in the evolution of Alleywood's service inventory was the application of service governance patterns, such as Service Decomposition (489), Proxy Capability (497), and Distributed Capability (510), which helped augment the Employee service to such an extent that a new Employee Records service was spawned with little disruption to the remaining inventory services.

Forestry Regulatory Commission (FRC)

The FRC's SOA effort was championed by their newly hired CTO who, along with a new set of directors, assumed control over a vast, legacy-ridden IT enterprise. The adoption of SOA was viewed as the wholesale solution to overcoming the skyrocketing operational costs that had been consuming more and more of the FRC's budget over the past years.

Subsequent to defining the planned service inventory, various FRC project teams went about modeling, designing, and building services. Because so much legacy logic needed encapsulation, the Legacy Wrapper (441) example typified a common service implementation, whereby outdated or proprietary APIs and interfaces needed to be wrapped by standardized service contracts. Other legacy encapsulation patterns, such as Multi-Channel Endpoint (451), helped further bring legacy assets into the service inventory so that they could be made available to a wider variety of consumers without imposing negative coupling requirements.

A key part of the overall SOA initiative was also establishing governance controls so that the planned inventory of services could be more effectively and efficiently maintained and evolved. This is where patterns such as Version Identification (472), Compatible Change (465), and Termination Notification (478) played important roles.

During its initial adoption stages, the FRC invested heavily in infrastructure upgrades which resulted in highly sophisticated messaging frameworks supported by the application of patterns such as Asynchronous Queuing (582), Intermediate Routing (549), Event-Driven Messaging (599), and Reliable Messaging (592). This allowed for the development of mission-critical solutions (e.g., the Flight Plan Validation service) that could leverage the quality-of-service features provided by the messaging framework, as well as the loosely coupled nature of message-based communication itself.

Combined with the architectural enhancements brought about by patterns such as Stateful Services (248) and Service Data Replication (350), the FRC has built an inventory of services that are scalable, reliable, and repeatedly composable into a variety of sophisticated solutions.

Appendix B



Candidate Patterns

The patterns documented in this book were in development for nearly 40 months prior to publication in printed format. During this period, the pattern catalog underwent exhaustive reviews by SOA experts and practitioners, key members of the patterns community, as well as members of the vendor and academic communities. Additionally, an open public review of the first draft of the manuscript of this book was held for several months at SOAPatterns.org during which 234 individual reviews were collected. The *History* and *Acknowledgements* pages at this Web site provide further details regarding past review stages and cycles.

The feedback collected as a result of this review process helped validate many patterns, but also helped identify those patterns that were either invalid or not yet ready to be considered fully proven or field-tested. At the time of this writing, a subset of the original patterns were deemed “not ready” for inclusion in the official SOA design pattern catalog and were therefore classified as *candidate patterns*.

Draft versions of these patterns are published in the *Candidate Patterns* part of the SOAPatterns.org Web site, where they are made available for public review until such a point that they are considered fully validated. After that, they are moved to the master SOA design patterns catalog.

Some notable patterns that were classified as candidates are those inspired by REST. A set of patterns co-authored by IBM’s Raj Balasubramanian (also co-author of the upcoming series title *SOA with REST*) was originally developed for this book but then excluded, as per recommendations from industry experts in the REST community. There was concern over the maturity of the REST-related technologies and concepts upon which the patterns were based. These patterns are expected to remain candidates for at least a year before they are individually re-assessed.

Other candidate patterns include several grid-related patterns contributed by David Chappell from Oracle and the Service Virtualization pattern from Satadru Roy of Sun Microsystems.

You are encouraged to visit SOAPatterns.org to view these and other candidate patterns and to also provide your comments and opinions using the online feedback forms. Note that SOAPatterns.org is different from SOAPatterns.com. The former is a community site dedicated to SOA patterns in general, whereas the latter acts as the official Web site for this book.

Appendix C



Principles of Service-Orientation

The first draft of the *SOA Design Patterns* manuscript was originally written for a book that was to be called “SOA: Principles and Patterns.” After the manuscript became too bulky for one book, it was decided to split it up into two titles: one about design patterns and one about design principles.

The latter was published in a book called *SOA Principles of Service Design*, which preceded *SOA Design Patterns*. Eight specific design principles were documented in this book, each explored in a dedicated chapter. Collectively, these principles comprise the service-orientation design paradigm, as explained earlier in Chapters 3 and 4.

As much as patterns inter-relate, principles too can relate to patterns as well as to each other. You may have noticed that the profile tables for the patterns in this book reference these principles wherever common relationships might exist. A cross-reference of these relationships is provided in Appendix D.

As an additional reference resource, the original profile tables for the principles themselves are provided in this appendix, as follows:

- Standardized Service Contract (Table C.1)
- Service Loose Coupling (Table C.2)
- Service Abstraction (Table C.3)
- Service Reusability (Table C.4)
- Service Autonomy (Table C.5)
- Service Statelessness (Table C.6)
- Service Discoverability (Table C.7)
- Service Composability (Table C.8)

You are also encouraged to visit SOAPrinciples.com for more background information about service-orientation.

NOTE

The profile tables in this appendix contain excerpts from *SOA Principles of Service Design* and therefore also include some terms and references specific to that book.

Standardized Service Contract	
Short Definition	<i>"Services share standardized contracts."</i>
Long Definition	<i>"Services within the same service inventory are in compliance with the same contract design standards."</i>
Goals	<ul style="list-style-type: none">• To enable services with a meaningful level of natural interoperability within the boundary of a service inventory. This reduces the need for data transformation because consistent data models are used for information exchange.• To allow the purpose and capabilities of services to be more easily and intuitively understood. The consistency with which service functionality is expressed through service contracts increases interpretability and the overall predictability of service endpoints throughout a service inventory. <p>Note that these goals are further supported by other service-orientation principles as well.</p>
Design Characteristics	<ul style="list-style-type: none">• A service contract (comprised of a technical interface or one or more service description documents) is provided with the service.• The service contract is standardized through the application of design standards.
Implementation Requirements	<p>The fact that contracts need to be standardized can introduce significant implementation requirements to organizations that do not have a history of using standards.</p> <p>For example:</p> <ul style="list-style-type: none">• Design standards and conventions need to ideally be in place prior to the delivery of any service in order to ensure adequately scoped standardization. (For those organizations that have already produced ad-hoc Web services, retro-fitting strategies may need to be employed.)• Formal processes need to be introduced to ensure that services are modeled and designed consistently, incorporating accepted design principles, conventions, and standards.

- Because achieving standardized Web service contracts generally requires a “contract first” approach to service-oriented design, the full application of this principle will often demand the use of development tools capable of importing a customized service contract without imposing changes.
- Appropriate skill-sets are required to carry out the modeling and design processes with the chosen tools. When working with Web services, the need for a high level of proficiency with XML schema and WSDL languages is practically unavoidable. WS-Policy expertise may also be required.

These and other requirements can add up to a noticeable transition effort that goes well beyond technology adoption.

Web Service Region of Influence

Because this principle is focused solely on the content of the service contract, its influence is limited to the contract and related processing logic within a typical Web service.

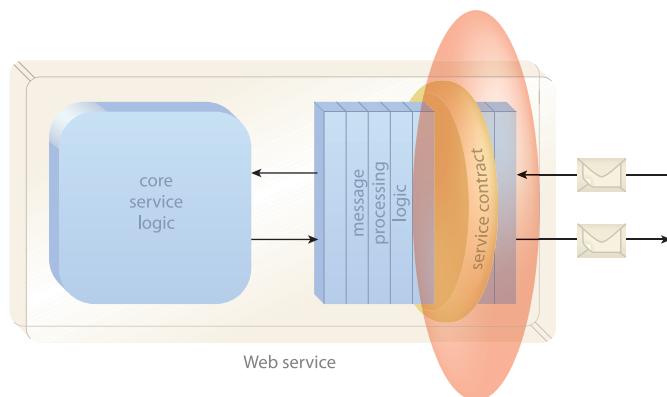


Table C.1

A profile for the Standardized Service Contract principle.

Service Loose Coupling	
Short Definition	<i>"Services are loosely coupled."</i>
Long Definition	<i>"Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment."</i>
Goals	By consistently fostering reduced coupling within and between services we are working toward a state where service contracts increase independence from their implementations and services are increasingly independent from each other. This promotes an environment in which services and their consumers can be adaptively evolved over time with minimal impact on each other.
Design Characteristics	<ul style="list-style-type: none">• The existence of a service contract that is ideally decoupled from technology and implementation details.• A functional service context that is not dependent on outside logic.• Minimal consumer coupling requirements.
Implementation Requirements	<ul style="list-style-type: none">• Loosely coupled services are typically required to perform more runtime processing than if they were more tightly coupled. As a result, data exchange in general can consume more runtime resources, especially during concurrent access and high usage scenarios.• To achieve the right balance of coupling, while also supporting the other service-orientation principles that affect contract design, requires increased service contract design proficiency.

Web Service Region of Influence

As we explore different coupling types in the next section, it will become evident that applying this principle touches numerous parts of the typical Web service architecture. However, the primary focal point, both for internal and consumer-related design considerations, remains the service contract.

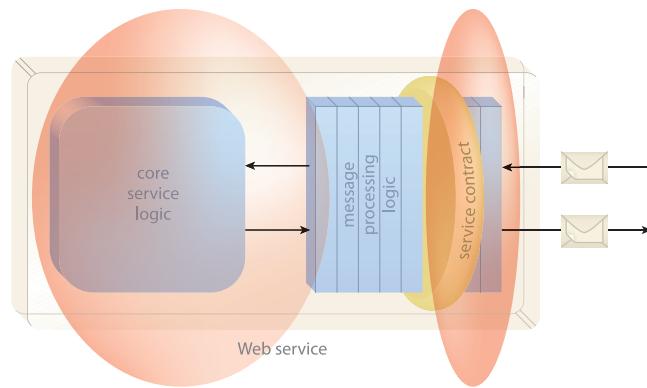


Table C.2

A profile for the Service Loose Coupling principle.

Service Abstraction	
Short Definition	<i>"Non-essential service information is abstracted."</i>
Long Definition	<i>"Service contracts only contain essential information and information about services is limited to what is published in service contracts."</i>
Goals	Many of the other principles emphasize the need to publish <i>more</i> information in the service contract. The primary role of this principle is to keep the quantity and detail of contract content concise and balanced and prevent unnecessary access to additional service details.
Design Characteristics	<ul style="list-style-type: none">Services consistently abstract specific information about technology, logic, and function away from the outside world (the world outside of the service boundary).Services have contracts that concisely define interaction requirements and constraints and other required service meta details.Outside of what is documented in the service contract, information about a service is controlled or altogether hidden within a particular environment.
Implementation Requirements	The primary prerequisite to achieving the appropriate level of abstraction for each service is the level of service contract design skill applied.
Web Service Region of Influence	The <i>Region of Influence</i> part of this profile has been moved to the <i>Types of Meta Abstraction</i> section (in the book <i>SOA Principles of Service Design</i>) where a separate Web service figure is provided for each form of abstraction.

Table C.3

A profile for the Service Abstraction principle.

Service Reusability	
Short Definition	<i>"Services are reusable."</i>
Long Definition	<i>"Services contain and express agnostic logic and can be positioned as reusable enterprise resources."</i>
Goals	<p>The goals behind Service Reusability are tied directly to some of the most strategic objectives of service-oriented computing:</p> <ul style="list-style-type: none">• To allow for service logic to be repeatedly leveraged over time so as to achieve an increasingly high return on the initial investment of delivering the service.• To increase business agility on an organizational level by enabling the rapid fulfillment of future business automation requirements through wide-scale service composition.• To enable the realization of agnostic service models.• To enable the creation of service inventories with a high percentage of agnostic services.
Design Characteristics	<ul style="list-style-type: none">• <i>The service is defined by an agnostic functional context</i>—The logic encapsulated by the service is associated with a context that is sufficiently agnostic to any one usage scenario so as to be considered reusable.• <i>The service logic is highly generic</i>—The logic encapsulated by the service is sufficiently generic, allowing it to facilitate numerous usage scenarios by different types of service consumers.• <i>The service has a generic and extensible contract</i>—The service contract is flexible enough to process a range of input and output messages.• <i>The service logic can be accessed concurrently</i>—Services are designed to facilitate simultaneous access by multiple consumer programs.
Implementation Requirements	<p>From an implementation perspective, Service Reusability can be the most demanding of the principles we've covered so far. Below are common requirements for creating reusable services and supporting their long-term existence:</p>

- A scalable runtime hosting environment capable of high-to-extreme concurrent service usage. Once a service inventory is relatively mature, reusable services will find themselves in an increasingly large number of compositions.
- A solid version control system to properly evolve contracts representing reusable services.
- Service analysts and designers with a high degree of subject matter expertise who can ensure that the service boundary and contract accurately represent the service's reusable functional context.
- A high level of service development and commercial software development expertise so as to structure the underlying logic into generic and potentially decomposable components and routines.

These and other requirements place an emphasis on the appropriate staffing of the service delivery team, as well as the importance of a powerful and scalable hosting environment and supporting infrastructure.

Web Service Region of Influence

This principle can affect all parts of a Web service. Contract design, the use of system messaging agents, and the underlying core logic can all be shaped by a service's reusability requirements.

When we view the service as an IT asset that requires an investment but provides the potential for repeated returns, we can appreciate why more care needs to be taken when designing each part of the service architecture.

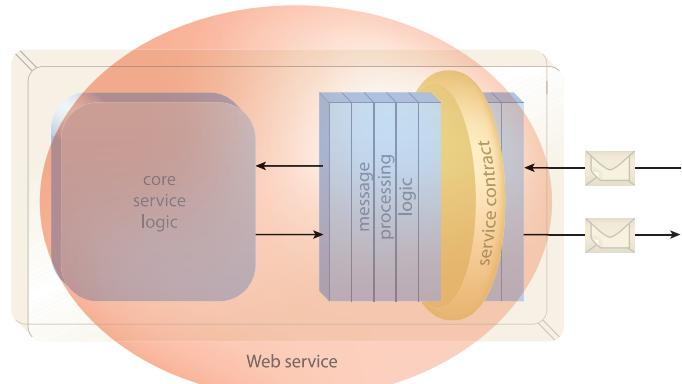


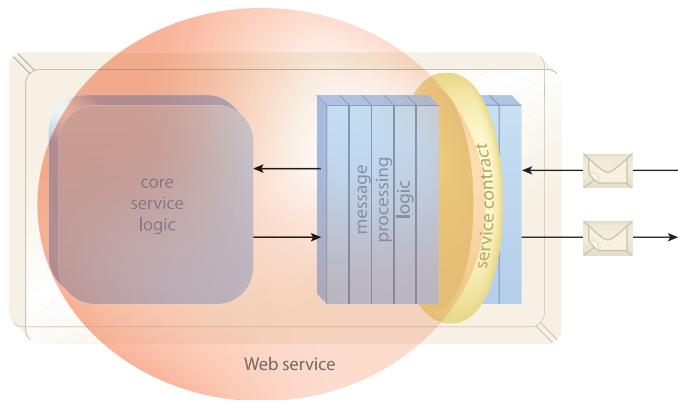
Table C.4

A profile for the Service Reusability principle.

Service Autonomy	
Short Definition	<i>"Services are autonomous."</i>
Long Definition	<i>"Services exercise a high level of control over their underlying runtime execution environment."</i>
Goals	<ul style="list-style-type: none">• To increase a service's runtime reliability, performance, and predictability, especially when being reused and composed.• To increase the amount of control a service has over its runtime environment. <p>By pursuing autonomous design and runtime environments, we are essentially aiming to increase post-implementation control over the service and the service's control over its own execution environment.</p>
Design Characteristics	<ul style="list-style-type: none">• Services have a contract that expresses a well-defined functional boundary that should not overlap with other services.• Services are deployed in an environment over which they exercise a great deal (and preferably an exclusive level) of control.• Service instances are hosted by an environment that accommodates high concurrency for scalability purposes.
Implementation Requirements	<ul style="list-style-type: none">• A high level of control over how service logic is designed and developed. Depending on the level of autonomy being sought, this may also involve control over the supporting data models.• A distributable deployment environment, so as to allow the service to be moved, isolated, or composed as required.• An infrastructure capable of supporting desired autonomy levels.

Web Service Region of Influence

Service Autonomy is almost exclusively focused on the service implementation, with an emphasis on the core service logic and any resources it may need at runtime. However, the service contract is also affected due to normalization considerations (as explained later).

**Table C.5**

A profile for the Service Autonomy principle.

Service Statelessness	
Short Definition	<i>"Services minimize statefulness."</i>
Long Definition	<i>"Services minimize resource consumption by deferring the management of state information when necessary."</i>
Goals	<ul style="list-style-type: none">• To increase service scalability.• To support the design of agnostic service logic and improve the potential for service reuse.
Design Characteristics	<p>What makes this somewhat of a unique principle is the fact that it is promoting a condition of the service that is temporary in nature. Depending on the service model and state deferral approach used, different types of design characteristics can be implemented. Some examples include:</p> <ul style="list-style-type: none">• Highly business process-agnostic logic so that the service is not designed to retain state information for any specific parent business process.• Less constrained service contracts so as to allow for the receipt and transmission of a wider range of state data at runtime.• Increased amounts of interpretative programming routines capable of parsing a range of state information delivered by messages and responding to a range of corresponding action requests.
Implementation Requirements	<p>Although state deferral can reduce the overall consumption of memory and system resources, services designed with statelessness considerations can also introduce some performance demands associated with the runtime retrieval and interpretation of deferred state data.</p> <p>Here is a short checklist of common requirements that can be used to assess the support of stateless service designs by vendor technologies and target deployment locations:</p> <ul style="list-style-type: none">• The runtime environment should allow for a service to transition from an idle state to an active processing state in a highly efficient manner.

- Enterprise-level or high-performance XML parsers and hardware accelerators (and SOAP processors) should be provided to allow services implemented as Web services to more efficiently parse larger message payloads with less performance constraints.
- The use of attachments may need to be supported by Web services to allow for messages to include bodies of payload data that do not undergo interface-level validation or translation to local formats.

The nature of the implementation support required by the average stateless service in an environment will depend on the state deferral approach used within the service-oriented architecture.

Web Service Region of Influence

Building a service to maximize the stateless condition affects the service contract design but can also directly influence how service logic is designed, right down to the individual programming routines and even the core algorithms that lie beneath each service capability.

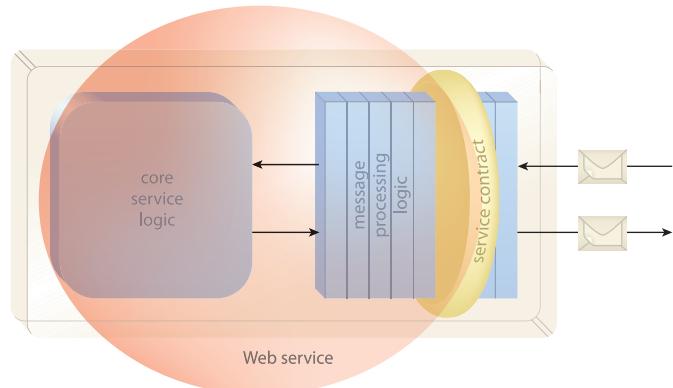


Table C.6

A profile for the Service Statelessness principle.

Service Discoverability	
Short Definition	<i>"Services are discoverable."</i>
Long Definition	<i>"Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted."</i>
Goals	<ul style="list-style-type: none">• Services are positioned as highly discoverable resources within the enterprise.• The purpose and capabilities of each service are clearly expressed so that they can be interpreted by humans and software programs. <p>Achieving these goals requires foresight and a solid understanding of the nature of the service itself. Depending on the type of service model being designed, realizing this principle may require both business and technical expertise.</p>
Design Characteristics	<ul style="list-style-type: none">• Service contracts are equipped with appropriate meta data that will be correctly referenced when discovery queries are issued.• Service contracts are further outfitted with additional meta information that clearly communicates their purpose and capabilities to humans.• If a service registry exists, registry records are populated with the same attention to meta information as just described.• If a service registry does not exist, service profile documents are authored to supplement the service contract and to form the basis for future registry records. (See Chapter 15 in <i>SOA Principles of Service Design</i> for more details about service profiles.)

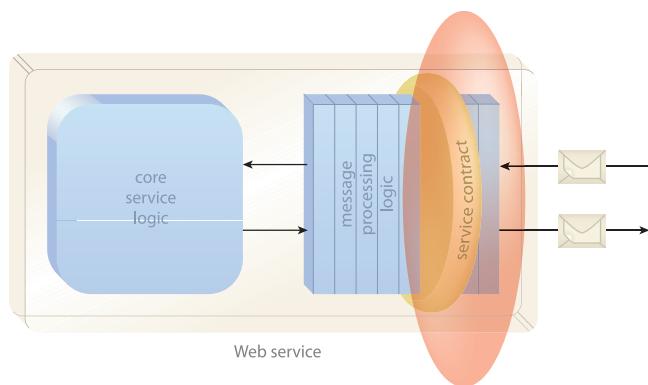
Implementation Requirements	<ul style="list-style-type: none">The existence of design standards that govern the meta information used to make service contracts discoverable and interpretable, as well as guidelines for how and when service contracts should be further supplemented with annotations.The existence of design standards that establish a consistent means of recording service meta information outside of the contract. This information is either collected in a supplemental document in preparation for a service registry, or it is placed in the registry itself. <p>You may have noticed the absence of a service registry on the list of implementation requirements. As previously established, the goal of this principle is to implement design characteristics within the service, not within the architecture.</p>
Web Service Region of Influence	<p>Even though we ultimately want a discovery mechanism in place, it is also ideal for service contracts to be independently discoverable and interpretable. From a Web service perspective, this principle is focused solely on the service contract documents.</p> 

Table C.7

A profile for the Service Discoverability principle.

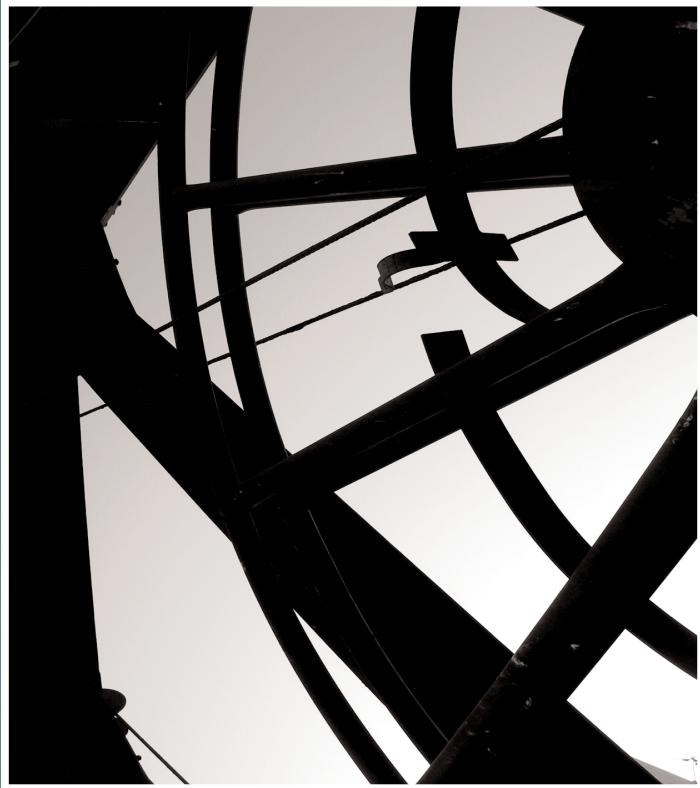
Service Composability	
Short Definition	<i>"Services are composable."</i>
Long Definition	<i>"Services are effective composition participants, regardless of the size and complexity of the composition."</i>
Goals	<p>When discussing the goals of Service Composability, pretty much all of the goals of Service Reusability apply. This is because service composition often turns out to be a form of service reuse. In fact, you may recall that one of the objectives we listed for the Service Reusability principle was to enable wide-scale service composition.</p> <p>However, above and beyond simply attaining reuse, service composition provides the medium through which we can achieve what is often classified as the ultimate goal of service-oriented computing. By establishing an enterprise comprised of solution logic represented by an inventory of highly reusable services, we provide the means for a large extent of future business automation requirements to be fulfilled through...you guessed it: service composition.</p>
Design Characteristics for Composition Member Capabilities	<p>Ideally, every service capability (especially those providing reusable logic) is considered a potential composition member. This essentially means that the design characteristics already established by the Service Reusability principle are equally relevant to building effective composition members.</p> <p>Additionally, there are two further characteristics emphasized by this principle:</p> <ul style="list-style-type: none">• The service needs to possess a highly efficient execution environment. More so than being able to manage concurrency, the efficiency with which composition members perform their individual processing should be highly tuned.• The service contract needs to be flexible so that it can facilitate different types of data exchange requirements for similar functions. This typically relates to the ability of the contract to exchange the same type of data at different levels of granularity.

	<p>The manner in which these qualities go beyond mere reuse has to do primarily with the service being capable of optimizing its runtime processing responsibilities in support of multiple, simultaneous compositions.</p>
Design Characteristics for Composition Controller Capabilities	<p>Composition members will often also need to act as controllers or sub-controllers within different composition configurations. However, services designed as designated controllers are generally alleviated from many of the high-performance demands placed on composition members.</p> <p>These types of services therefore have their own set of design characteristics:</p> <ul style="list-style-type: none">• The logic encapsulated by a designated controller will almost always be limited to a single business task. Typically, the task service model is used, resulting in the common characteristics of that model being applied to this type of service.• While designated controllers may be reusable, service reuse is not usually a primary design consideration. Therefore, the design characteristics fostered by Service Reusability are considered and applied where appropriate, but with less of the usual rigor applied to agnostic services.• Statelessness is not always as strictly emphasized on designated controllers as with composition members. Depending on the state deferral options available by the surrounding architecture, designated controllers may sometimes need to be designed to remain fully stateful while the underlying composition members carry out their respective parts of the overall task. <p>Of course, any capability acting as a controller can become a member of a larger composition, which brings the previously listed composition member design characteristics into account as well.</p>

Table C.8

A profile for the Service Composability principle.

This page intentionally left blank

A black and white abstract photograph featuring a complex arrangement of thick, dark, intersecting lines forming a grid-like structure. The lines create various shapes, including triangles and rectangles, against a lighter background. The overall effect is one of geometric abstraction and visual complexity.

Appendix D

Patterns and Principles Cross-Reference

For quick reference purposes, this appendix provides a master cross-reference of service-orientation design principles and SOA design patterns based on the principles listed in the *Principles* cell of the profile tables that begin each pattern description. Note that these design principles are briefly explained in Appendix C and more information about the service-orientation design paradigm is available at SOAPrinciples.com.

Design Principle	Referenced by Patterns
Standardized Service Contract	<ul style="list-style-type: none">Agnostic Capability (324)Asynchronous Queuing (582)Canonical Expression (275)Canonical Protocol (150)Canonical Schema (158)Canonical Versioning (286)Capability Composition (521)Capability Recomposition (526)Compatible Change (465)Concurrent Contracts (421)Contract Centralization (409)Contract Denormalization (414)Data Format Transformation (681)Data Model Transformation (671)Decomposed Capability (504)Decoupled Contract (401)Distributed Capability (510)Domain Inventory (123)

Design Principle	Referenced by Patterns
Standardized Service Contract	<p>Dual Protocols (227)</p> <p>Enterprise Inventory (116)</p> <p>Event-Driven Messaging (599)</p> <p>Inventory Endpoint (260)</p> <p>Legacy Wrapper (441)</p> <p>Message Screening (381)</p> <p>Non-Agnostic Context (319)</p> <p>Partial Validation (362)</p> <p>Policy Centralization (207)</p> <p>Protocol Bridging (687)</p> <p>Schema Centralization (200)</p> <p>Service Callback (566)</p> <p>Service Façade (333)</p> <p>Service Messaging (533)</p> <p>Service Refactoring (484)</p> <p>State Messaging (557)</p> <p>Termination Notification (478)</p> <p>Validation Abstraction (429)</p> <p>Version Identification (472)</p>
Service Loose Coupling	<p>Asynchronous Queuing (582)</p> <p>Capability Composition (521)</p> <p>Capability Recomposition (526)</p> <p>Compatible Change (465)</p> <p>Compensating Service Transaction (631)</p> <p>Concurrent Contracts (421)</p>

(continues)

(continued)

Design Principle	Referenced by Patterns
Service Loose Coupling	<ul style="list-style-type: none">Contract Centralization (409)Contract Denormalization (414)Data Format Transformation (681)Decoupled Contract (401)Dual Protocols (227)Entity Abstraction (175)Event-Driven Messaging (599)File Gateway (457)Intermediate Routing (549)Inventory Endpoint (260)Legacy Wrapper (441)Messaging Metadata (538)Multi-Channel Endpoint (451)Partial Validation (362)Policy Centralization (207)Process Abstraction (182)Proxy Capability (497)Schema Centralization (200)Service Agent (543)Service Callback (566)Service Decomposition (489)Service Façade (333)Service Instance Routing (574)Service Messaging (533)Service Perimeter Guard (394)Service Refactoring (484)

Design Principle	Referenced by Patterns
Service Loose Coupling	Trusted Subsystem (387) UI Mediator (366) Utility Abstraction (168) Validation Abstraction (429)
Service Abstraction	Capability Composition (521) Capability Recomposition (526) Decomposed Capability (504) Domain Inventory (123) Dual Protocols (227) Enterprise Inventory (116) Entity Abstraction (175) Exception Shielding (376) Inventory Endpoint (260) Legacy Wrapper (441) Policy Centralization (207) Process Abstraction (182) Service Perimeter Guard (394) Service Refactoring (484) Utility Abstraction (168) Validation Abstraction (429)
Service Reusability	Agnostic Capability (324) Agnostic Context (312) Agnostic Sub-Controller (607) Capability Composition (521) Capability Recomposition (526)

(continues)

(continued)

Design Principle	Referenced by Patterns
Service Reusability	Composition Autonomy (616) Concurrent Contracts (421) Cross-Domain Utility Layer (267) Data Model Transformation (671) Entity Abstraction (175) Intermediate Routing (549) Logic Centralization (136) Multi-Channel Endpoint (451) Rules Centralization (216) Service Agent (543) Service Layers (143) Utility Abstraction (168)
Service Autonomy	Canonical Resources (237) Capability Composition (521) Capability Recomposition (526) Composition Autonomy (616) Distributed Capability (510) Dual Protocols (227) Event-Driven Messaging (599) Process Centralization (193) Redundant Implementation (345) Service Data Replication (350) Service Normalization (131)

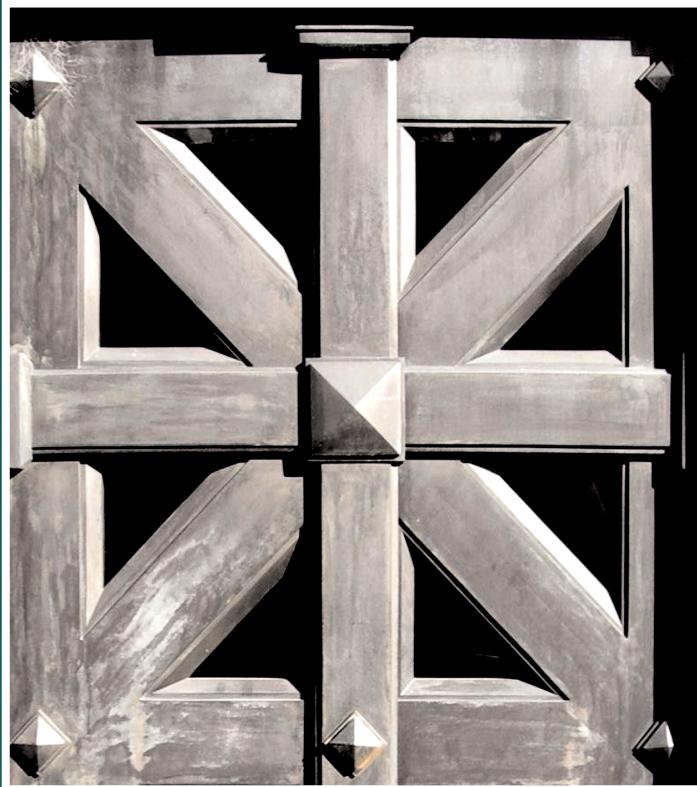
Design Principle	Referenced by Patterns
Service Statelessness	Asynchronous Queuing (582) Atomic Service Transaction (623) Capability Composition (521) Capability Recomposition (526) Messaging Metadata (538) Partial State Deferral (356) Process Centralization (193) Service Grid (254) Service Instance Routing (574) State Messaging (557) State Repository (242) Stateful Services (248)
Service Discoverability	Canonical Expression (275) Capability Composition (521) Capability Recomposition (526) Metadata Centralization (280)
Service Composability	Agnostic Capability (324) Agnostic Sub-Controller (607) Brokered Authentication (661) Capability Composition (521) Capability Recomposition (526) Composition Autonomy (616) Cross-Domain Utility Layer (267) Data Confidentiality (641) Data Model Transformation (671)

(continues)

(continued)

Design Principle	Referenced by Patterns
Service Composability	<ul style="list-style-type: none">Data Origin Authentication (649)Direct Authentication (656)Domain Inventory (123)Dual Protocols (227)Enterprise Inventory (116)Entity Abstraction (175)Intermediate Routing (549)Logic Centralization (136)Non-Agnostic Context (319)Process Abstraction (182)Process Centralization (193)Protocol Bridging (687)Reliable Messaging (592)Service Callback (566)Service Decomposition (489)Service Instance Routing (574)Service Layers (143)State Messaging (557)Utility Abstraction (168)

Appendix E



Patterns and Architecture Types Cross-Reference

Provided in this appendix is a cross-reference table with the four SOA architecture types established in Chapter 4 referenced against SOA design patterns, based on the architecture types listed in the *Architecture* cell of each pattern profile table.

Architecture Type	Referenced by Patterns
Service	<ul style="list-style-type: none">Agnostic Capability (324)Agnostic Context (312)Agnostic Sub-Controller (607)Brokered Authentication (661)Canonical Expression (275)Canonical Protocol (150)Canonical Schema (158)Canonical Versioning (286)Capability Composition (521)Capability Recomposition (526)Compatible Change (465)Concurrent Contracts (421)Contract Denormalization (414)Data Confidentiality (641)Data Format Transformation (681)Decomposed Capability (504)Decoupled Contract (401)Direct Authentication (656)Distributed Capability (510)

Architecture Type	Referenced by Patterns
Service	<p>Dual Protocols (227)</p> <p>Entity Abstraction (175)</p> <p>Event-Driven Messaging (599)</p> <p>Exception Shielding (376)</p> <p>File Gateway (457)</p> <p>Functional Decomposition (300)</p> <p>Legacy Wrapper (441)</p> <p>Logic Centralization (136)</p> <p>Message Screening (381)</p> <p>Multi-Channel Endpoint (451)</p> <p>Non-Agnostic Context (319)</p> <p>Partial State Deferral (356)</p> <p>Policy Centralization (193)</p> <p>Process Abstraction (182)</p> <p>Proxy Capability (497)</p> <p>Redundant Implementation (345)</p> <p>Schema Centralization (200)</p> <p>Service Callback (566)</p> <p>Service Data Replication (350)</p> <p>Service Decomposition (489)</p> <p>Service Encapsulation (305)</p> <p>Service Façade (333)</p> <p>Service Grid (254)</p> <p>Service Instance Routing (574)</p> <p>Service Layers (143)</p> <p>Service Messaging (533)</p>

continues

(continued)

Architecture Type	Referenced by Patterns
Service	Service Normalization (131) Service Perimeter Guard (394) Service Refactoring (484) State Messaging (557) State Repository (242) Stateful Services (248) Termination Notification (478) Trusted Subsystem (387) Utility Abstraction (168) Validation Abstraction (429) Version Identification (472)
Composition	Agnostic Sub-Controller (607) Asynchronous Queuing (582) Atomic Service Transaction (623) Brokered Authentication (661) Capability Composition (521) Capability Recomposition (526) Compensating Service Transaction (631) Composition Autonomy (616) Contract Centralization (409) Data Confidentiality (641) Data Format Transformation (681) Data Model Transformation (671) Data Origin Authentication (649) Direct Authentication (656)

Architecture Type	Referenced by Patterns
Composition	<ul style="list-style-type: none">Entity Abstraction (175)Intermediate Routing (549)Logic Centralization (136)Messaging Metadata (538)Partial Validation (362)Process Abstraction (182)Process Centralization (193)Reliable Messaging (592)Service Agent (543)Service Callback (566)Service Instance Routing (574)Service Messaging (533)State Messaging (557)Termination Notification (478)UI Mediator (366)Utility Abstraction (168)
Inventory	<ul style="list-style-type: none">Asynchronous Queuing (582)Atomic Service Transaction (623)Brokered Authentication (661)Canonical Expression (275)Canonical Protocol (150)Canonical Resources (237)Canonical Schema (158)Canonical Versioning (286)Capability Composition (521)

continues

(continued)

Architecture Type	Referenced by Patterns
Inventory	Capability Recomposition (526) Compensating Service Transaction (631) Cross-Domain Utility Layer (267) Data Confidentiality (641) Data Format Transformation (681) Data Model Transformation (671) Domain Inventory (123) Enterprise Inventory (116) Logic Centralization (136) Service Normalization (131) Dual Protocols (227) Entity Abstraction (175) Event-Driven Messaging (599) Inventory Endpoint (260) Metadata Centralization (280) Partial State Deferral (356) Policy Centralization (207) Process Abstraction (182) Process Centralization (193) Reliable Messaging (592) Rules Centralization (216) Schema Centralization (200) Service Agent (543) Service Callback (566) Service Data Replication (350)

Architecture Type	Referenced by Patterns
Inventory	Service Grid (254) Service Instance Routing (574) Service Layers (143) Service Messaging (533) State Repository (242) Stateful Services (248) Utility Abstraction (168)
Enterprise	Canonical Expression (275) Canonical Resources (237) Cross-Domain Utility Layer (267) Domain Inventory (123) Enterprise Inventory (116) Metadata Centralization (280) Service Grid (254)

Table E.1

This page intentionally left blank

About the Author

Thomas Erl is the world's top-selling SOA author, Series Editor of the *Prentice Hall Service-Oriented Computing Series from Thomas Erl*, and Editor of *The SOA Magazine* (www.soamag.com). With over 100,000 copies in print world-wide, his books have become international bestsellers and have been formally endorsed by senior members of major software organizations, such as IBM, Microsoft, Oracle, BEA, Sun, Intel, SAP, CISCO, and HP.

His most recent titles *SOA Design Patterns* and *Web Service Contract Design and Versioning for SOA* were co-authored with a series of industry experts and follow his first three books *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*, *Service-Oriented Architecture: Concepts, Technology, and Design*, and *SOA Principles of Service Design*.

Thomas is currently working with over 20 authors on the upcoming titles: *SOA Governance*, *SOA with .NET*, *SOA with Java*, *ESB Architecture for SOA*, and *SOA with REST*. He is also overseeing the SOAPatterns.org initiative, a community site dedicated to SOA patterns.

Thomas is the founder of SOA Systems Inc. (www.soasystems.com), a company specializing in vendor-neutral SOA consulting and training services. Thomas is also the founder of the internationally recognized SOA Certified Professional program (www.soacp.com and www.soaschool.com). Thomas is a speaker and instructor for private and public events and is regularly invited to Gartner summits. He has delivered many workshops and keynote speeches, and is on the program committee for the International SOA Symposium. Articles and interviews by Thomas have been published in numerous publications, including *The Wall Street Journal* and *CIO Magazine*.

For more information, visit: www.thomaserl.com.

About the Contributors

Provided in this section are biographies for some of the contributors. For a cross-reference of contributors and patterns, see the *Index of Patterns* page located before the main index at the end of this book.

David Chappell

David Chappell is Vice President and Chief Technologist for SOA at Oracle, where he is driving the vision for Oracle's SOA Grid initiative. David has more thanr 20 years of experience in the software industry. He is well known worldwide for his writings and public lectures on the subjects of service-oriented architecture (SOA), the enterprise service bus (ESB), message oriented middleware (MOM), enterprise integration, and is a co-author of many advanced Web Services standards. As author of *Enterprise Service Bus* (O'Reilly, 2004), David has had tremendous impact on redefining the shape and definition of SOA infrastructure. David is also currently working on a separate title dedicated to grid-enabled service-oriented architecture, a topic he has already written extensively about.

Kevlin Henney

Kevlin Henney is an independent consultant and trainer based in the UK. His areas of interest and practice are software architecture and patterns, programming languages and techniques, and agile development processes and practices. He is and has been a columnist for a number of software development magazines and sites, including Better Software, The Register, Java Report, and C++ Report. Kevlin is co-author of Volumes 4 and 5 in the *Pattern-Oriented Software Architecture series* (A Pattern Language for Distributed Computing and On Patterns and Pattern Languages).

Florent Georges

Florent is a freelance IT consultant in Brussels who has been involved in the XML world for more than 11 years and is a recognized expert within the XSLT and XQuery communities. Soon after discovering Web services, he had the opportunity to work with them for several years with major European institutions and companies in the financial and IT consultancy sectors in Brussels and London, such as ING and Atos Origin. In 2008 Florent won the SOACP contest by tying for the highest score among all participants in a series of certification workshops associated with the 2008 International SOA Symposium. As a result, Florent received his SOA Architect certification with a special “Top Gun” designation. Florent maintains a blog at www.fgeorges.org.

Jason Hogg

Jason Hogg is an Architect inside the Microsoft Services Managed Solutions Group. Jason has been with Microsoft for six years where he has also worked inside Microsoft’s Patterns & Practices and Microsoft Research divisions. While at Microsoft Jason has specialized in topics relating to the design of distributed applications, with an emphasis on SOA, security, and interoperability. Prior to joining Microsoft Jason, worked for 12 years as a consultant in the United States, Great Britain, and Australia at organizations including The United Nations, WorldNow, J Sainsbury’s, British Airways, and BankWest . Jason holds an MSc (CompSci) from the University of Washington.

Anish Karmarkar

Anish Karmarkar, Ph.D., is a Consulting Member of Technical Staff at Oracle and is part of the standards and strategy team responsible for SOA, Web services, and Java specifications. He has 17 years of research, development, and standards experience in various aspects of distributed systems and protocols. Anish is a co-editor of various Web services standards, including SOAP 1.2, WS-ReliableMessaging, WS-ReliableMessaging Policy, WS-Make-Connection, WS-I Basic Profile, WS-I Reliable Secure Profile, WS-I Attachments Profile, RRSHB, amongst others. He is also a co-editor of the SCA set of specifications. As an active participant and a founding member in various Web services and SOA-related Working Groups, Technical Committees, Expert Groups in W3C, OASIS, JCP, and OSOA collaboration, he has played a significant role in the development of Web services and SCA standards and specifications. Anish has been on the Board of Directors of the OSGi Alliance since 2006, co-chair of the OASIS SCA BPEL Technical Committee, and Oracle’s alternate representative on the JCP “Big Java” Executive Committee. He has also served as the vice-Chair of WS-I Basic Profile Working Group. He received his Ph.D. in Computer Science from Texas A&M University in 1997.

Mark Little

Dr. Mark Little is Engineering Director at Red Hat where he is the Technical Development Manager for the JBoss SOA Platform. Prior to this he was the lead on the JBossESB and JBossTS projects, where he is still the Development Manager. Mark has over 20 years of experience in using and developing distributed systems, that includes being Chief Architect at Arjuna Technologies and a Distinguished Engineer at Hewlett Packard, where he lead the transactions team to produce the world's first Web Services transactions product. Mark spends a lot of time working in various standards bodies, including OASIS and W3C, and has co-authored several of the WS-* standards. He has presented regularly at conferences, workshops and events such as JavaOne, HP World, JBoss World, the SOA Symposium, and the World Wide Web Conference where he has also been a co-chair on the Web Services track. Mark is also a co-author of a number of books including *Java Transaction Processing* and the upcoming title *ESB Architecture for SOA*.

Brian Lokhorst MSc, BSc, SOACP

Brian Lokhorst is a Solution Architect for the new Declarations Management System (DMS) of the Dutch Tax and Customs Administration (DTCA). Previously, Brian worked as an integration-technology architect in the program responsible for getting the DTCA transformed into a service oriented organization. He advised on the gradual migration towards a service-oriented way of thinking, with a special focus on Web services and the use of ESB-technology for its technical implementation. As technical team lead and senior infrastructure developer Brian was responsible for a B2B-integration implemented with Web services and ESB-technology between two government agencies. Prior to that he was responsible for developing the ICT service portfolio architecture and the architecture of software factories for the DTCA. Brian is a Certified SOA Architect and an IBM Certified RUP 7.1 Solution Designer. Brian holds an MSc in Business Economics & ICT from the University of Groningen and a BSc in International Business Economics from the School of Business and Economics at Windesheim University.

Brian Loesgen

Based in San Diego, Brian Loesgen is a Principal Consultant with Neudesic, a firm that specializes in .NET development and Microsoft server integration. Brian is a five-time Microsoft MVP for BizTalk Server and has extensive experience building sophisticated enterprise and mobile solutions. In addition, Brian has been involved with advanced Enterprise Service Bus solutions, and was a key architect and developer of the "Microsoft ESB Guidance" released by Microsoft in Oct 2006. He is a co-author of six books, including

BizTalk Server 2004 Unleashed, and is currently working on *SOA with .NET*. He has written technical white papers for Intel, Microsoft, and others and has spoken at numerous major technical conferences worldwide. Brian is a co-founder and past-President of the International .NET Association (ineta.org). He is the President of the San Diego .NET user group, leads the San Diego Software Industry Council SOA SIG, and is a member of the Editorial Board for the .NET Developer's Journal. Brian is also a member of the Microsoft Connected Systems Division Virtual Technical Specialist Team, and is part of Microsoft's Connected Systems Advisory Board. Brian's blog is at blog.BrianLoesgen.com.

Berthold Maier

Berthold Maier works for Oracle Consulting Germany as Chief Architect and has more than 15 years experience as a developer, coach, and architect building complex, mission-critical applications and integration architectures. Within the last seven years at Oracle he has held several positions in the German consulting organization with a focus on Java/JEE, Integration, SOA, BPM, DB, ECM, security, and, most recently, enterprise architecture management.

Since December 2006 Berthold has been the acting Chief Architect for the entire Oracle consulting division in Germany. In this position he is responsible for reference architectures pertaining to integration and JEE, and for the Oracle Consulting Development Frameworks. He is also the originator and architect of the Accelerate Consulting Framework based on MDA, SOA and ADF. Berthold is a well-known speaker and author, and is also a co-founder of the Masons-of-SOA, an inter-company network founded by architects of Oracle Germany, Opitz, SOPERA (Eclipse Project Swordfish founders), and EDS.

Hajo Normann

Hajo Normann is an SOA/BPM evangelist and architect at EDS. His interest for business focused, enterprise-wide, cross-siloed bundles of functionality (services) first emerged in 2001 while acting as the architect and technical team lead for a shared service platform at a large German bank. In 2003, Hajo modeled business processes at the Federal Office and is now working on a governmental project focused on BPMN models and executable WS-BPEL processes. Much of Hajo's research has been concentrated on the missing links between SOA and BPM and he is regularly engaged in discussions and thought exchange with BPM and ESB thought leaders at Oracle Headquarters. Hajo co-leads the German DOAG SIG SOA, co-founded the Masons-of-SOA, acts as an Oracle ACE Director, and is a frequent speaker at public events.

Chris Riley, SOACP

Chris Riley is a Senior Consultant and Instructor with SOA Systems Inc. and has been part of many engagements across North America. In his capacity as a consultant, Chris is working with Fortune 500 organizations (such as Pearson, Kodak, SAIC, and State of Indiana) in the financial services, consumer products, education, and government sectors in their pursuit of enterprise SOA. He specializes in helping organizations move from traditional distributed computing processes to SOA using open standards. Prior to SOA Systems, Chris ran the Professional Service department for Cape Clear Software as Director of Architecture helping North American customers implement Enterprise Service Bus technology. In this role, Chris also coordinated and supported webinars and speaking engagements promoting ESB and SOA with Gartner and Forrester, as well as public forums, such as the Integration Consortium, Toronto Java User Group and SOA Seminar Series. Previous to Cape Clear, Chris worked at Extricity, Inc. as a Senior Solution Consultant and PTC as a manager of Technical Services.

Thomas Rischbeck

Thomas is an IT architect and business developer with the Swiss-based [ipt] consultancy group. He has many years of experience in the delivery of complex e-business architectures in the government and logistics domains. Thomas advises clients on their enterprise architecture and works on various technical and organizational topics with a focus on SOA infrastructure and SOA governance. Thomas has a strong foundation in asynchronous integration middleware and distributed and parallel architectures. Prior to joining [ipt], Thomas worked as a senior engineer with the Hewlett Packard middleware division (ex Bluestone). At HP he was the lead developer for the JMS message broker and its integration into the HP application server. After the HP-Compaq merger in 2001, Thomas took on the role of solutions architect with Arjuna Ltd. Thomas holds a B.Sc. in Computing Science and a Ph.D. in Parallel Computing, both from the University of Newcastle upon Tyne. He is a frequent speaker and also active as an author for the upcoming book *ESB Architecture for SOA*.

Satadru Roy

Satadru Roy is a Senior SOA Architect with Sun Microsystems, Canada where he consults with clients to provide architecture and design guidance for their SOA initiatives. Satadru has extensive consulting experience in various domain verticals and he has also worked for well-known infrastructure software vendors, such as BEA Systems, Inc. He has extensive experience in middleware, application integration and BPM, and is a strong proponent of

open source-based integration technologies. Satadru is one of the co-authors of the upcoming *SOA with Java* title.

Arnaud Simon

Arnaud Simon is a Red Hat senior middleware consultant and co-author of the upcoming book *ESB Architecture for SOA*. Prior to Red Hat Arnaud was an IT Architect working for IPT, a European systems integrator based in Switzerland. Before that he was leading the Message Service Team for Arjuna Technologies Limited, a spin-off from Hewlett-Packard where he was a senior research and development engineer. Arnaud obtained his Ph.D in Computing Science from INRIA and completed research posts at both INRIA and Newcastle University.

Bernd Trops

Bernd Trops is the Director of Professional Services at SOPERA GmbH. Since the mid-90s he worked for several companies (including GemStone, Brokat, WebGain, and Oracle) as a Systems Engineer and coach on countless OO and J2EE projects.

During his Oracle days Bernd held the position of SOA Architect with focus on large-scale initiatives. One notable project included the Deutsche Posts Service Back Bone, which eventually became the foundation for the Eclipse SOA Runtime Framework's core Project Swordfish. Today he is responsible for consulting and training at SOPERA, which supports an OpenSource SOA platform contributed to the Eclipse Foundation. Bernd is also a co-founder of the Masons-of-SOA.

Clemens Utschig-Utschig

An eight year Oracle veteran, Clemens works for the SOA Product Management team at Oracle Headquarters in the USA where he is responsible for product strategy and developer adoption. During the early days of Java Clemens founded the local Java Community and years later he was nominated as Austria's representative into the EMEA Java/XML community. Clemens spent years consulting in Switzerland and Dubai and performed workshops all around Europe. In 2004 he was the first consultant from EMEA to be invited for an internship with Oracle where he spent a month developing the XML Datacontrol for the JSR-227 reference-implementation that is since part of the official samples. Since his transfer into the SOA/integration engineering group, Clemens is responsible for cross product integration and currently acts as advisor for SOA to Fusion Applications development. He is an advisory member of the Applications Architecture Board and a member of

the OASIS SCA assembly Technical Committee. In 2006 Clemens was designated as the youngest Oracle ACE for contributions to the developer community on OTN, and graduated the same year as the youngest Certified Project Manager from Stanford University. He is a frequent speaker at conferences and has published articles in known industry journals. Clemens is an advisory member of Oracle's Arch2Arch community, serves in the advisory board for SOA at the Germany-based JAX/SOACON conference, and is co-founder of the Masons-of-SOA.

Torsten Winterberg

Torsten Winterberg works for the Oracle Advantage Partner OPITZ Consulting as the director of the Application Development division. He is a long-time developer, coach, and architect with specializations in the areas of Java EE, BPM, BPEL, ESB, BAM and SOA in general. He is a known speaker in the German Java community and has written numerous articles on SOA-related topics. He is currently focused on the design and architecture aspects of projects based on the Oracle SOA Suite. Torsten is part of the Masons of SOA and is co-authoring an article series on "Hot SOA topics not tackled yet by others" for the German Javamagazin. He is also a member of the advisory board for two big German conferences (Jax, W-Jax), was BPM track chair for Jax 2008, and helps lead the DOAG Special Interest Group SOA. Torsten was recognized by Oracle for his evangelist role by being the designated Oracle ACE Director since 2007.

Dennis Wisnosky

Dennis Wisnosky is Chief Technical Officer of the US Department of Defense (DoD) Business Mission Area within the office of the Deputy Under Secretary of Defense for Business Transformation (OUSD (BT)). He is recognized as a creator of the Integrated Definition language, the standard for modeling and analysis in management and business improvement efforts. Wisnosky holds a bachelor's degree in physics and mathematics from California University of Pennsylvania, a master's in management science from the University of Dayton, and a master's in electrical engineering from the University of Pittsburgh.

Index of Patterns

- Agnostic Capability (Erl), 324
- Agnostic Context (Erl), 312
- Agnostic Sub-Controller (Erl), 607
- Asynchronous Queuing (Little, Rischbeck, Simon), 582
- Atomic Service Transaction (Erl), 623
- Brokered Authentication (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham), 661
- Canonical Expression (Erl), 275
- Canonical Protocol (Erl), 150
- Canonical Resources (Erl), 237
- Canonical Schema (Erl), 158
- Canonical Schema Bus (Utschig, Maier, Trops, Normann, Winterberg, Erl), 709
- Canonical Versioning (Erl), 286
- Capability Composition (Erl), 521
- Capability Recomposition (Erl), 526
- Compatible Change (Orchard, Riley), 465
- Compensating Service Transaction (Utschig, Maier, Trops, Normann, Winterberg, Loesgen, Little), 631
- Composition Autonomy (Erl), 616
- Concurrent Contracts (Erl), 421
- Contract Centralization (Erl), 409
- Contract Denormalization (Erl), 414
- Cross-Domain Utility Layer (Erl), 267
- Data Confidentiality (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado,

- Taylor, Wall, Slater, Imran, Cibraro, Cunningham), 641
- Data Format Transformation (Little, Rischbeck, Simon), 681
- Data Model Transformation (Erl), 671
- Data Origin Authentication (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham), 649
- Decomposed Capability (Erl), 504
- Decoupled Contract (Erl), 401
- Direct Authentication (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham), 656
- Distributed Capability (Erl), 510
- Domain Inventory (Erl), 123
- Dual Protocols (Erl), 227
- Enterprise Inventory (Erl), 116
- Enterprise Service Bus (Erl, Little, Rischbeck, Simon), 704
- Entity Abstraction (Erl), 175
- Event-Driven Messaging (Little, Rischbeck, Simon), 599
- Exception Shielding (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham), 376
- Federated Endpoint Layer (Erl), 713
- File Gateway (Roy), 457
- Functional Decomposition (Erl), 300
- Intermediate Routing (Little, Rischbeck, Simon), 549
- Inventory Endpoint (Erl), 260
- Legacy Wrapper (Erl, Roy), 441
- Logic Centralization (Erl), 136
- Message Screening (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham), 381
- Messaging Metadata (Erl), 538
- Metadata Centralization (Erl), 280
- Multi-Channel Endpoint (Roy), 451
- Non-Agnostic Context (Erl), 319
- Official Endpoint (Erl), 711
- Orchestration (Erl, Loesgen), 701

- Partial State Deferral (Erl), 356
- Partial Validation (Orchard, Riley), 362
- Policy Centralization (Erl), 207
- Process Abstraction (Erl), 182
- Process Centralization (Erl), 193
- Protocol Bridging (Little, Rischbeck, Simon), 687
- Proxy Capability (Erl), 497
- Redundant Implementation (Erl), 345
- Reliable Messaging (Little, Rischbeck, Simon), 592
- Rules Centralization (Erl), 216
- Schema Centralization (Erl), 200
- Service Agent (Erl), 543
- Service Broker (Little, Rischbeck, Simon), 707
- Service Callback (Karmarkar), 566
- Service Data Replication (Erl), 350
- Service Decomposition (Erl), 489
- Service Encapsulation (Erl), 305
- Service Façade (Erl), 333
- Service Grid (Chappell), 254
- Service Instance Routing (Karmarkar), 574
- Service Layers (Erl), 143
- Service Messaging (Erl), 533
- Service Normalization (Erl), 131
- Service Perimeter Guard (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham), 394
- Service Refactoring (Erl), 484
- State Messaging (Karmarkar), 557
- State Repository (Erl), 242
- Stateful Services (Erl), 248
- Termination Notification (Orchard, Riley), 478
- Three-Layer Inventory (Erl), 715
- Trusted Subsystem (Hogg, Smith, Chong, Hollander, Kozaczynski, Brader, Delgado, Taylor, Wall, Slater, Imran, Cibraro, Cunningham), 387

UI Mediator (Utschig, Maier, Trops, Normann, Winterberg), 366

Utility Abstraction (Erl), 168

Validation Abstraction (Erl), 429

Version Identification (Orchard, Riley), 472

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*
 See SOA
 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

A

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*
- 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
SOAPrinciples.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 also 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.thomasrl.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

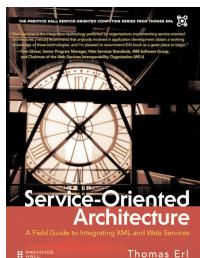
XML Schema language, 206, 752

XML Schema, spelling, 206

XML-Encryption, 640, 643

XML-Signature, 640, 652

XSLT, 674



Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services

ISBN 0131428985

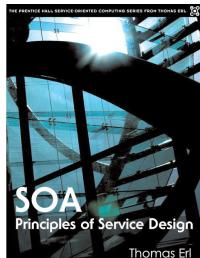
This top-selling field guide offers expert advice for incorporating XML and Web services technologies within service-oriented integration architectures.



Service-Oriented Architecture: Concepts, Technology, and Design

ISBN 0131858580

Widely regarded as the definitive “how-to” guide for SOA, this best-selling book presents a comprehensive end-to-end tutorial that provides step-by-step instructions for modeling and designing service-oriented solutions from the ground up.



SOA Principles of Service Design

ISBN 0132344823

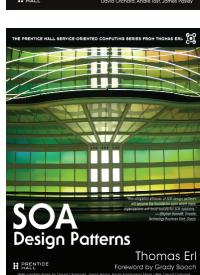
Published with over 240 color illustrations, this hands-on guide contains practical, comprehensive, and in-depth coverage of service engineering techniques and the service-orientation design paradigm. Proven design principles are documented to help maximize the strategic benefit potential of SOA.



Web Service Contract Design and Versioning for SOA

ISBN: 9780136135173

For Web services to succeed as part of SOA, they require balanced, effective technical contracts that enable services to be evolved and repeatedly reused for years to come. Now, a team of industry experts presents the first end-to-end guide to designing and governing Web service contracts.



SOA Design Patterns

ISBN 0136135161

Software design patterns have emerged as a powerful means of avoiding and overcoming common design problems and challenges. This new book presents a formal catalog of design patterns specifically for SOA and service-orientation. All patterns are documented using full-color illustrations and further supplemented with case study examples.

Several additional series titles are currently in development and will be released soon. For more information about any of the books in this series, visit www.soabooks.com.



REGISTER



THIS PRODUCT

informit.com/register

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account.

You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

informIT.com

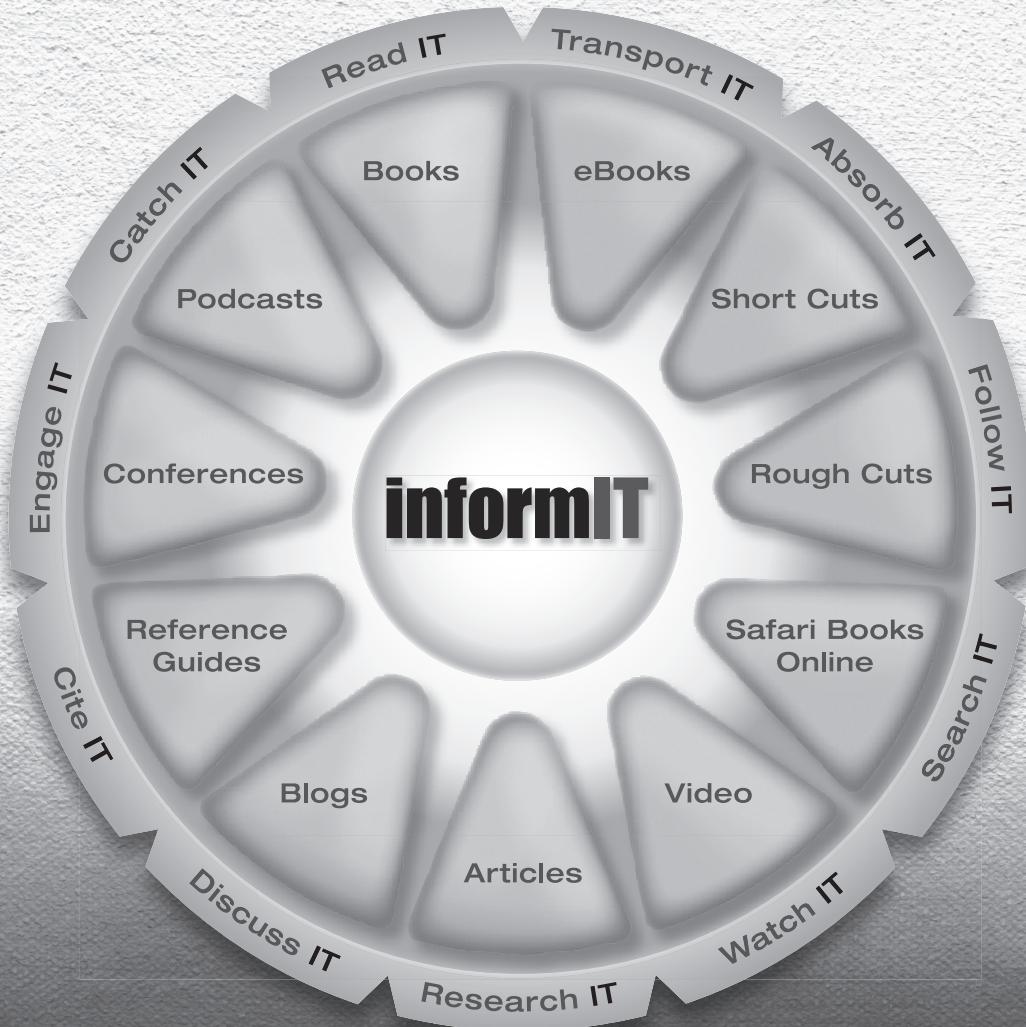
THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE

LearnIT at InformIT

Go Beyond the Book



11 WAYS TO LEARN IT at www.informIT.com/learn

The digital network for the publishing imprints of Pearson Education

Try Safari Books Online FREE

Get online access to 5,000+ Books and Videos



FREE TRIAL—GET STARTED TODAY!
www.informit.com/safaritrial

Find trusted answers, fast

Only Safari lets you search across thousands of best-selling books from the top technology publishers, including Addison-Wesley Professional, Cisco Press, O'Reilly, Prentice Hall, Que, and Sams.

Master the latest tools and techniques

In addition to gaining access to an incredible inventory of technical books, Safari's extensive collection of video tutorials lets you learn from the leading video training experts.

WAIT, THERE'S MORE!

Keep your competitive edge

With Rough Cuts, get access to the developing manuscript and be among the first to learn the newest technologies.

Stay current with emerging technologies

Short Cuts and Quick Reference Sheets are short, concise, focused content created to get you up-to-speed quickly on new and cutting-edge technologies.



AdobePress



Cisco Press

FT Press
FINANCIAL TIMES

IBM
Press



Microsoft
Press



O'REILLY



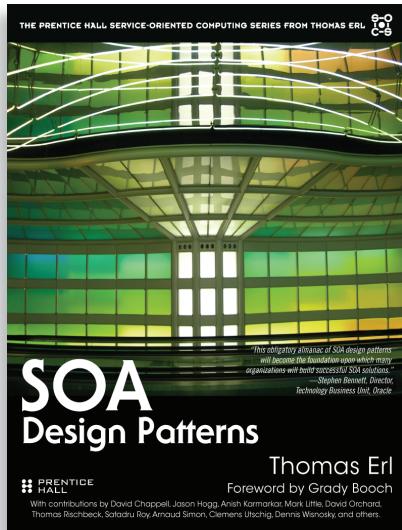
QUE



SAMS

SAS
Publishing





FREE Online Edition

Your purchase of **SOA Design Patterns** includes access to a free online edition for 45 days through the Safari Books Online subscription service. Nearly every Prentice Hall book is available online through Safari Books Online, along with more than 5,000 other technical books and videos from publishers such as Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, O'Reilly, Que, and Sams.

SAFARI BOOKS ONLINE allows you to search for a specific answer, cut and paste code, download chapters, and stay current with emerging technologies.

**Activate your FREE Online Edition at
www.informit.com/safarifree**

➤ **STEP 1:** Enter the coupon code: CIVUHXA.

➤ **STEP 2:** New Safari users, complete the brief registration form.
Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition,
please e-mail customer-service@safaribooksonline.com

Safari
Books Online

Addison
Wesley

AdobePress

ALPHA

Cisco Press

FT Press
FINANCIAL TIMES

IBM
Press

lynda.com

Microsoft
Press

New
Riders

O'REILLY

Peachpit
Press

PRENTICE
HALL

que®

Redbooks

SAMS

Sas
Publishing

Sun
microsystems

WILEY
Wiley
School
Publishing

WILEY

The SOA Magazine

The SOA Magazine is a monthly online publication provided by SOA Systems Inc. and Prentice Hall, and is officially associated with the *Prentice Hall Service-Oriented Computing Series from Thomas Erl*.

The SOA Magazine is dedicated to publishing specialized SOA articles, case studies, and papers by industry experts and professionals. The common criteria for contributions is that each explore a distinct aspect of service-oriented computing.

Visit The SOA Magazine at www.soamag.com or www.soamagazine.com. If you are interested in contributing, use the online form. If you would like to be automatically notified when new issues are published, send a blank e-mail to: notify@soasystems.com



The International SOA Symposium

The International SOA Symposium is the world's most comprehensive SOA event for practitioners, showcasing the leading SOA experts and speakers from around the world. The theme of the event is "substance only", with an emphasis on ensuring that each session provides in-depth coverage and true educational value for the most important SOA-related topics, including:

- SOA Architecture & Design
- Service Modeling & BPM
- SOA & Business
- SOA & REST
- SOA & Web 2.0
- SOA Governance
- SOA Programming
- SOA Innovations
- SOA Infrastructure & Technology
- SOA Project Delivery & Methodology

Additionally, the SOA Symposium regularly includes:

- Book Launch Ceremonies
- Expert Panels
- Hundreds of Free Books for Contests & Giveaways
- Exclusive Content

The International SOA Symposium attracts the world's leading SOA experts, including many of the authors of the *Prentice Hall Service-Oriented Computing Series from Thomas Erl*.

For more information, visit www.soasymposium.com. To be notified of the latest event information, send a blank e-mail to: notify@soasymposium.com