

Praise for This Book

“An absolute pleasure to read...the best SOA book I’ve read.

A book I would recommend to all of my colleagues; it provides much insight to the topics often overlooked by most books in this genre...the visuals were fantastic.”

—*Brandon Bohling, SOA Architecture and Strategy, Intel Corporation*

“I recommend this book to any SOA practitioner who wishes to empower themselves in making service design real...gives readers the 360° view into service design [and] gives SOA practitioners the depth and understanding needed into the principles of SOA to assist in the design of a mature and successful SOA program.”

—*Stephen G. Bennett, Americas SOA Practice Lead, BEA Systems*

“There are few references for SOA that give you the nuts and bolts and this one is at the top of the list. Well written and valuable as a reference book to any SOA practitioner.”

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

“A very clear discussion of the subject matter. Provides a good structure that facilitates understanding and readily highlights key points.”

—*Kareem Yusuf, Director of SOA Strategy and Planning, IBM Software Group*

“This book does a great job laying out benefits, key ideas and design principles behind successfully adopting service-oriented computing. At the same time, the book openly addresses challenges, risks and trade-offs that are in the way of adopting SOA in the real-world today. It moves away from ivory-tower views of service orientation, but still lays out a strong vision for SOA and outlines the changes necessary to realize the full potential.”

—*Christoph Schittko, Senior Architect, Microsoft*

“This book strikes a healthy balance between theory and practice. It is a perfect complement to the SOA series by the author.”

—*Prakash Narayan, Sun Microsystems*

“This book could be described as an encyclopedia of service design—Erl leaves nothing to chance. Indispensable.”

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

“I liked this book. It contains extremely important material for those who need to design services.”

—*Farzin Yashar, IBM SOA Advanced Technologies*

“Thomas Erl’s books are always densely filled with information that’s well structured. This book is especially insightful for Enterprise Architects because it provides great context and practical examples. Part 1 of the book alone is worth getting the book for.”

—*Markus Zirn, Senior Director, Product Management,
Oracle Fusion Middleware, Oracle Corporation*

“This book is a milestone in SOA literature. For the first time we are provided with a practical guide on defining service characteristics and service design principles for SOA from a vendor-agnostic viewpoint. It’s a great reference for SOA discovery, adoptions, and implementation projects.”

—*Canyang Kevin Liu, Principal Enterprise Architect, SAP Americas, Inc.*

“There are very few who understand SOA like Thomas Erl does! The principle centric description of service orientation from Thomas canonizes the underpinnings of this important paradigm shift in creating agile and reusable software capabilities. The principles, so eloquently explained, leave little room for any ambiguity attached to the greater purpose of SOA. Most organizations today are creating services in a bottoms-up approach, realizing composition and reuse organically. The time is ripe for a book like this that prepares architects for a principle centric approach to SOA.”

—*Hanu Kommalapati, Architect, Microsoft Corporation*

“If you are going to be designing, developing, or implementing SOA, this is a must have book.”

—*Jason “AJ” Comfort Sr., Booz Allen Hamilton*

“An excellent book for anyone who wants to understand service-orientation and the principles involved in designing services...a clear, concise and articulate exploration of the eight design principles involved in analyzing, designing, implementing, and maintaining services...”

—*Anish Karmarkar, Oracle Corporation*

“Very well written, succinct, and easy to understand.”

—*Raj Balasubramanian, IBM Software Group*

“A thorough examination of the considerations of service design. Both seasoned SOA practitioners and those endeavoring to realize services can benefit from reading this book.”

—*Bill Draven, Enterprise Architect, Intel Corporation*

“I am very impressed. Comprehensive. Educative. This book helped me to step back and look at the SOA principles from broader perspective. I’d say this is a must-read book for SOA stakeholders.”

—*Radovan Janecek, Director R&D, SOA Center, Hewlett-Packard*

“A comprehensive exploration of the issues of service design which has the potential to become the definitive work in this area.”

—*James Pasley, Chief Technology Officer, Cape Clear Software*

“SOA projects are most successful when they are based on a solid technical foundation. Well accepted and established design principles are part of this foundation. This book takes a very structured approach at defining the core design principles for SOA, thus allowing the reader to immediately applying them to a project. Each principle is formally introduced and explained, and examples are given for how to apply it to a real design problem. A ‘must read’ for any architect, designer or developer of service oriented solutions”.

—*Andre Tost, Senior Technical Staff Member, IBM Software Group*

“Outstanding SOA literature uniquely focused on the fundamental services design with thorough and in-depth study on all practical aspects from design principles to methodologies. This book provides a systematic approach for SOA adoption essential for both IT management and professionals.”

—Robin Chen, PhD, Google, Inc.

“An excellent addition to any SOA library; it covers a wide range of issues in enough detail to be a valuable asset to anyone considering designing or using SOA based technologies.”

—Mark Little, Director of Standards, Red Hat

“Very valuable guidance for understanding and applying SOA service design principles with concrete examples. A must read for the practitioner of SOA service design.”

—Umit Yalcinalp, PhD, Standards Architect, SAP

“This book communicates complex concepts in a clear and concise manner. Examples and illustrations are used very effectively.”

—Darryl Hogan, Senior Architect, Microsoft

“This book really does an excellent job of explaining the principles underpinning the value of SOA...Erl goes to great length to explain and give examples of each of the 8 principles that will significantly increase the readers ability to drive an SOA service design that benefits both business and IT.”

—Robert Laird, IT Architect, IBM EAI/SOA Advanced Technologies Group

“A work of genius...Offers the most comprehensive and thorough explanation on the principles of service design and what it means to be ‘service oriented.’

“Erl’s treatment of the complex world of service oriented architecture is pragmatic, inclusive of real world situations and offers readers ways to communicate these ideas through illustrations and well formulated processes.”

—David Michalowicz, MITRE Corporation

“This is the book for the large organization trying to rationalize its IT assets and establish an agile platform for the future. By highlighting risk and rewards, Thomas Erl brings clarity to how Service Orientation can be applied to ensure a responsive IT organization. This book finally brings software engineering principles to address the real world development challenges being faced.

To effectively serve the business, let alone embrace SOA, everyone involved should be familiar with the concepts investigated here. Thomas Erl thoroughly clarifies the nuances and defines the practice of service design.

We expect that this will become a classic text in software engineering, corporate training and colleges.”

—Cory Isaacson, *President, Rogue Wave Software and Ravi Palepu, SOA Author and Speaker*

“Thomas Erl does a great job...an easy read.”

—Michael H. Sor, *Booz Allen Hamilton*

“...a must read for SOA Architects to develop a firm foundation and understanding of the principles (and trade-offs) that make up a good SOA service.

After reading this book, it finally ‘clicked’ as to why a properly designed SOA system is different (and better) than a system based on previous enterprise architectures.”

—Fred Ingham, *Platinum Solutions Inc.*

“Lays a tremendous foundation for business and technical workers to come to common terms and expectations...incredibly enlightening to see the details associated with achieving the SOA vision.”

—Wayne P. Ariola, *Vice President of Strategy, Parasoft*

“[Erl does] and excellent job of addressing the breadth of [his] audience to present to those new to SOA and weaved in enough detail to assist those who are already actively involved in SOA development.”

—R. Perry Smith, *Application Program Manager, EDS/OnStar*

“It is easy to miss the big picture of what SOA means for the design of larger scale systems amidst the details of WS technologies. Erl helps provide a broader perspective, surveying the landscape from a design standpoint.”

—*Jim Clune, Chief Architect, Parasoft*

“Lays a firm foundation for the underlying principles of good service design. Cuts through the hype and provides a cogent resource for improving architectural judgment on SOA projects.”

—*Jim Murphy, Vice President of Product Management, Mindreef, Inc.*

“The first book to concisely, gradually and comprehensively explain how to apply SOA principles into enterprise-level software design. It is an excellent book.”

—*Robin G. Qiu, Ph.D., Division of Engineering and Information Science,
Pennsylvania State University*

“I really think that this is a very useful book that a lot of people really need out there in the industry.”

—*Dr. Arnaud Simon, Principal Software Engineer, Red Hat*

“...indispensable companion to designing and implementing a service-oriented architecture. It condenses all information necessary to design services and is the most relevant source I know if in the field.”

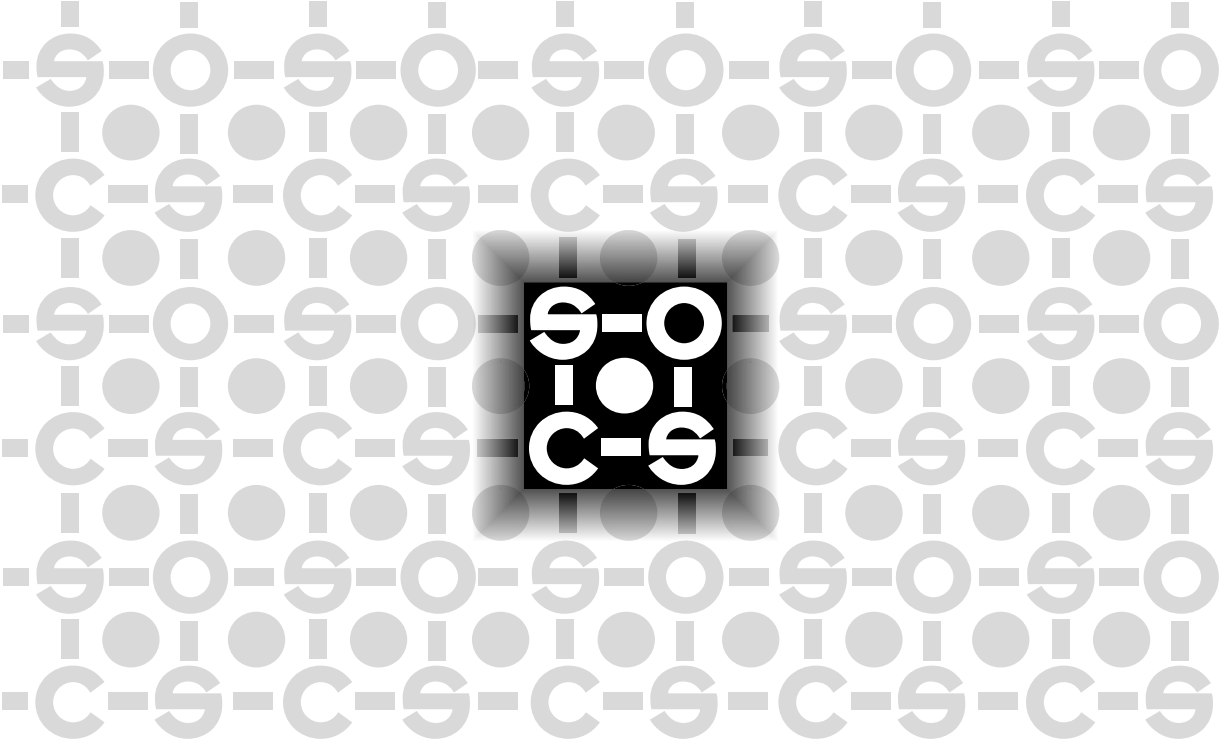
“[This book is] not only helpful, but fundamental to successfully designing an SOA.”

—*Phillipp Offermann, Research Analyst, University of Berlin*

“Service-Oriented Architecture is an important topic in IT today. Its vast scope could span an organization’s enterprise. Designing it properly is a major undertaking. This book provides timely, expert and comprehensive discussions on the principles of service design. Thomas has a keen sense in identifying the subtle points of various subjects and explains them in an easy-to-understand way. The book is a valuable resource for IT professionals working in SOA.”

—*Peter H. Chang, PhD, Associate Professor of Information Systems,
Lawrence Technological University*

SOA: Principles of Service Design



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

For more information, visit www.soabooks.com.

SOA

Principles of Service Design

Thomas Erl

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@pearsoned.com

Editor-in-Chief

Mark L. Taub

Managing Editor

Gina Kanouse

Senior Project Editor

Kristy Hart

Copy Editor

Language Logistics, LLC

Senior Indexer

Cheryl Lenser

Proofreader

Williams Woods Publishing

Publishing Coordinator

Noreen Regina

Composer

Jake McFarland

Cover Designer

Thomas Erl

Graphics

Zuzana Cappova
Spencer Fruhling

Photos

Thomas Erl

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.

Visit us on the Web: www.pearson.com

Library of Congress Cataloging-in-Publication Data

Erl, Thomas.

SOA: principles of service design / Thomas Erl.

p. cm.

ISBN 0-13-234482-3 (hardback : alk. paper) 1. Web services. 2. Computer architecture. 3. System analysis.

4. System design. I. Title.

TK5105.88813.E75 2008

004.2'2—dc22

Copyright © 2008 SOA Systems, Inc.

All photographs provided by Thomas Erl. Permission to use photographs granted by 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: 9780132344821

ISBN-10: 0132344823

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

Ninth Printing: May 2014

To my wife and family for their support.

This page intentionally left blank

Contents

Preface	xxv
----------------------	------------

Chapter 1: Introduction	1
--------------------------------------	----------

1.1 Objectives of this Book	3
1.2 Who this Book Is For	3
1.3 What this Book Does Not Cover	4
Topics Covered by Other Books	4
SOA Standardization Efforts	5
1.4 How this Book Is Organized	6
Part I: Fundamentals	7
Part II: Design Principles	9
Part III: Supplemental	12
Appendices	12
1.5 Symbols, Figures, and Style Conventions	13
Symbol Legend	13
How Color Is Used	13
The Service Symbol	13
1.6 Additional Information	16
Updates, Errata, and Resources (www.soabooks.com)	16
Master Glossary (www.soaglossary.com)	16
Referenced Specifications (www.soaspecs.com)	16
Service-Oriented Computing Poster (www.soaposters.com)	16

The SOA Magazine (www.soamag.com) 17
 Notification Service 17
 Contact the Author 17

Chapter 2: Case Study 19

2.1 Case Study Background: Cutit Saws Ltd. 20
 History 20
 Technical Infrastructure and Automation Environment 21
 Business Goals and Obstacles 21

PART I: FUNDAMENTALS

Chapter 3: Service-Oriented Computing and SOA. 25

3.1 Design Fundamentals 26
 Design Characteristic 27
 Design Principle 28
 Design Paradigm 29
 Design Pattern 30
 Design Pattern Language 31
 Design Standard 32
 Best Practice 34
 A Fundamental Design Framework 35
 3.2 Introduction to Service-Oriented Computing 37
 Service-Oriented Architecture 38
 Service-Oriented Architecture, Services, and Service-Oriented
 Solution Logic 39
 Service Compositions 39
 Service Inventory 40
 Understanding Service-Oriented Computing Elements 40
 Service Models 43
 SOA and Web Services 46
 Service Inventory Blueprints 51
 Service-Oriented Analysis and Service Modeling 52

- Service-Oriented Design 53
- Service-Oriented Architecture: Concepts, Technology,
and Design 54
- 3.3 Goals and Benefits of Service-Oriented Computing 55
 - Increased Intrinsic Interoperability 56
 - Increased Federation 58
 - Increased Vendor Diversification Options 59
 - Increased Business and Technology Domain Alignment 60
 - Increased ROI 61
 - Increased Organizational Agility 63
 - Reduced IT Burden 64
- 3.4 Case Study Background 66

Chapter 4: Service-Orientation 67

- 4.1 Introduction to Service-Orientation 68
 - Services in Business Automation 69
 - Services Are Collections of Capabilities 69
 - Service-Orientation as a Design Paradigm 70
 - Service-Orientation and Interoperability 74
- 4.2 Problems Solved by Service-Orientation 75
 - Life Before Service-Orientation 76
 - The Need for Service-Orientation 81
- 4.3 Challenges Introduced by Service-Orientation 85
 - Design Complexity 85
 - The Need for Design Standards 86
 - Top-Down Requirements 86
 - Counter-Agile Service Delivery in Support of Agile
Solution Delivery 87
 - Governance Demands 88
- 4.4 Additional Considerations 89
 - It Is Not a Revolutionary Paradigm 89
 - Enterprise-wide Standardization Is Not Required 89
 - Reuse Is Not an Absolute Requirement 90

- 4.5 Effects of Service-Orientation on the Enterprise 91
 - Service-Orientation and the Concept of “Application” 91
 - Service-Orientation and the Concept of “Integration”. 92
 - The Service Composition 94
 - Application, Integration, and Enterprise Architectures 95
- 4.6 Origins and Influences of Service-Orientation 96
 - Object-Orientation. 97
 - Web Services 98
 - Business Process Management (BPM) 98
 - Enterprise Application Integration (EAI) 98
 - Aspect-Oriented Programming (AOP) 99
- 4.7 Case Study Background 100

Chapter 5: Understanding Design Principles 103

- 5.1 Using Design Principles 104
 - Incorporate Principles within Service-Oriented Analysis 105
 - Incorporate Principles within Formal Design Processes. 106
 - Establish Supporting Design Standards. 107
 - Apply Principles to a Feasible Extent 108
- 5.2 Principle Profiles 109
- 5.3 Design Pattern References 111
- 5.4 Principles that Implement vs. Principles that Regulate. . . 111
- 5.5 Principles and Service Implementation Mediums. 114
 - “Capability” vs. “Operation” vs. “Method” 115
- 5.6 Principles and Design Granularity 115
 - Service Granularity 116
 - Capability Granularity 116
 - Data Granularity 116
 - Constraint Granularity 117
 - Sections on Granularity Levels 118
- 5.7 Case Study Background 119
 - The Lab Project Business Process 119

PART II: DESIGN PRINCIPLES

- Chapter 6: Service Contracts (Standardization and Design) 125**
- 6.1 Contracts Explained 126
 - Technical Contracts in Abstract 126
 - Origins of Service Contracts 127
- 6.2 Profiling this Principle 130
- 6.3 Types of Service Contract Standardization 132
 - Standardization of Functional Service Expression 133
 - Standardization of Service Data Representation 134
 - Standardization of Service Policies 137
- 6.4 Contracts and Service Design 140
 - Data Representation Standardization and Transformation Avoidance. 140
 - Standardization and Granularity 142
 - Standardized Service Contracts and Service Models 144
 - How Standardized Service Contract Design Affects Other Principles 144
- 6.5 Risks Associated with Service Contract Design 149
 - Versioning 149
 - Technology Dependencies 150
 - Development Tool Deficiencies 151
- 6.6 More About Service Contracts 152
 - Non-Technical Service Contract Documents 152
 - “Web Service Contract Design for SOA”. 153
- 6.7 Case Study Example 154
 - Planned Services 154
 - Design Standards 155
 - Standardized WSDL Definition Profiles 155
 - Standardized XML Schema Definitions 157
 - Standardized Service and Data Representation Layers 157
 - Service Descriptions 158
 - Conclusion 160

Chapter 7: Service Coupling (Intra-Service and Consumer Dependencies) 163

- 7.1 Coupling Explained 164
 - Coupling in Abstract 165
 - Origins of Software Coupling 165
- 7.2 Profiling this Principle 167
- 7.3 Service Contract Coupling Types 169
 - Logic-to-Contract Coupling (the coupling of service logic to the service contract) 173
 - Contract-to-Logic Coupling (the coupling of the service contract to its logic). 174
 - Contract-to-Technology Coupling (the coupling of the service contract to its underlying technology) 176
 - Contract-to-Implementation Coupling (the coupling of the service contract to its implementation environment). 177
 - Contract-to-Functional Coupling (the coupling of the service contract to external logic). 180
- 7.4 Service Consumer Coupling Types. 181
 - Consumer-to-Implementation Coupling 182
 - Standardized Service Coupling and Contract Centralization . . . 185
 - Consumer-to-Contract Coupling 185
 - Measuring Consumer Coupling 191
- 7.5 Service Loose Coupling and Service Design 193
 - Coupling and Service-Orientation. 193
 - Service Loose Coupling and Granularity 195
 - Coupling and Service Models. 196
 - How Service Loose Coupling Affects Other Principles 197
- 7.6 Risks Associated with Service Loose Coupling 200
 - Limitations of Logic-to-Contract Coupling 200
 - Problems when Schema Coupling Is “too loose” 201
- 7.7 Case Study Example. 202
 - Coupling Levels of Existing Services 202
 - Introducing the InvLegacyAPI Service 203
 - Service Design Options 205

Chapter 8: Service Abstraction (Information Hiding and Meta Abstraction Types) 211

- 8.1 Abstraction Explained 212
 - Origins of Information Hiding 213
- 8.2 Profiling this Principle 214
 - Why Service Abstraction Is Needed 214
- 8.3 Types of Meta Abstraction 218
 - Technology Information Abstraction 219
 - Functional Abstraction 221
 - Programmatic Logic Abstraction. 222
 - Quality of Service Abstraction. 224
 - Meta Abstraction Types and the Web Service Regions of Influence 225
 - Meta Abstraction Types in the Real World 227
- 8.4 Measuring Service Abstraction. 231
 - Contract Content Abstraction Levels 231
 - Access Control Levels. 232
 - Abstraction Levels and Quality of Service Meta Information . . . 234
- 8.5 Service Abstraction and Service Design 235
 - Service Abstraction vs. Service Encapsulation. 235
 - How Encapsulation Can Affect Abstraction 235
 - Service Abstraction and Non-Technical Contract Documents . . 237
 - Service Abstraction and Granularity 238
 - Service Abstraction and Service Models 239
 - How Service Abstraction Affects Other Principles 239
- 8.6 Risks Associated with Service Abstraction 242
 - Multi-Consumer Coupling Requirements 242
 - Misjudgment by Humans 242
 - Security and Privacy Concerns. 243
- 8.7 Case Study Example. 244
 - Service Abstraction Levels 244
 - Operation-Level Abstraction Examples 247

Chapter 9: Service Reusability (Commercial and Agnostic Design) 253

- 9.1 Reuse Explained 254
 - Reuse in Abstract 254
 - Origins of Reuse 257
- 9.2 Profiling this Principle 259
- 9.3 Measuring Service Reusability and Applying
 - Commercial Design. 262
 - Commercial Design Considerations 262
 - Measures of Planned Reuse 265
 - Measuring Actual Reuse 267
 - Commercial Design Versus Gold-Plating 267
- 9.4 Service Reuse in SOA 268
 - Reuse and the Agnostic Service 268
 - The Service Inventory Blueprint 269
- 9.5 Standardized Service Reuse and Logic Centralization . . 270
 - Understanding Logic Centralization 271
 - Logic Centralization as an Enterprise Standard 272
 - Logic Centralization and Contract Centralization 272
 - Centralization and Web Services 274
 - Challenges to Achieving Logic Centralization 274
- 9.6 Service Reusability and Service Design 276
 - Service Reusability and Service Modeling 276
 - Service Reusability and Granularity 277
 - Service Reusability and Service Models. 278
 - How Service Reusability Affects Other Principles 278
- 9.7 Risks Associated with Service Reusability and
 - Commercial Design. 281
 - Cultural Concerns 281
 - Governance Concerns 283
 - Reliability Concerns 286
 - Security Concerns. 286
 - Commercial Design Requirement Concerns. 286
 - Agile Delivery Concerns 287

- 9.8 Case Study Example 288
 - The Inventory Service Profile 288
 - Assessing Current Capabilities 289
 - Modeling for a Targeted Measure of Reusability 289
 - The New EditItemRecord Operation 290
 - The New ReportStockLevels Operation 290
 - The New AdjustItemsQuantity Operation 291
 - Revised Inventory Service Profile 292

Chapter 10: Service Autonomy (Processing Boundaries and Control) 293

- 10.1 Autonomy Explained 294
 - Autonomy in Abstract 294
 - Origins of Autonomy 295
- 10.2 Profiling this Principle 296
- 10.3 Types of Service Autonomy 297
 - Runtime Autonomy (execution) 298
 - Design-Time Autonomy (governance) 298
- 10.4 Measuring Service Autonomy 300
 - Service Contract Autonomy (services with normalized contracts) 301
 - Shared Autonomy 305
 - Service Logic Autonomy (partially isolated services) 306
 - Pure Autonomy (isolated services) 308
 - Services with Mixed Autonomy 310
- 10.5 Autonomy and Service Design 311
 - Service Autonomy and Service Modeling 311
 - Service Autonomy and Granularity 311
 - Service Autonomy and Service Models 312
 - How Service Autonomy Affects Other Principles 314
- 10.6 Risks Associated with Service Autonomy 317
 - Misjudging the Service Scope 317
 - Wrapper Services and Legacy Logic Encapsulation 318
 - Overestimating Service Demand 318

- 10.7 Case Study Example 319
 - Existing Implementation Autonomy of the GetItem Operation . . 319
 - New Operation-Level Architecture with Increased Autonomy . . 320
 - Effect on the Run Lab Project Composition 322

Chapter 11: Service Statelessness (State Management Deferral and Stateless Design) 325

- 11.1 State Management Explained 327
 - State Management in Abstract 327
 - Origins of State Management 328
 - Deferral vs. Delegation 331
- 11.2 Profiling this Principle 331
- 11.3 Types of State 335
 - Active and Passive 335
 - Stateless and Stateful 336
 - Session and Context Data 336
- 11.4 Measuring Service Statelessness 339
 - Non-Deferred State Management (low-to-no statelessness) . . 340
 - Partially Deferred Memory (reduced statefulness) 340
 - Partial Architectural State Management Deferral (moderate statelessness) 341
 - Full Architectural State Management Deferral (high statelessness) 342
 - Internally Deferred State Management (high statelessness) . . 342
- 11.5 Statelessness and Service Design 343
 - Messaging as a State Deferral Option 343
 - Service Statelessness and Service Instances 344
 - Service Statelessness and Granularity 346
 - Service Statelessness and Service Models 346
 - How Service Statelessness Affects Other Principles 347
- 11.6 Risks Associated with Service Statelessness 349
 - Dependency on the Architecture 349
 - Increased Runtime Performance Demands 350
 - Underestimating Delivery Effort 350

- 11.7 Case Study Example 351
 - Solution Architecture with State Management Deferral 352
 - Step 1 353
 - Step 2 354
 - Step 3 355
 - Step 4 356
 - Step 5 357
 - Step 6 358
 - Step 7 359

Chapter 12: Service Discoverability (Interpretability and Communication) 361

- 12.1 Discoverability Explained 362
 - Discovery and Interpretation, Discoverability and Interpretability in Abstract 364
 - Origins of Discovery 367
- 12.2 Profiling this Principle 368
- 12.3 Types of Discovery and Discoverability
 - Meta Information 371
 - Design-Time and Runtime Discovery 371
 - Discoverability Meta Information 373
 - Functional Meta Data 374
 - Quality of Service Meta Data 374
- 12.4 Measuring Service Discoverability 375
 - Fundamental Levels 375
 - Custom Rating System 376
- 12.5 Discoverability and Service Design 376
 - Service Discoverability and Service Modeling 377
 - Service Discoverability and Granularity 378
 - Service Discoverability and Policy Assertions 378
 - Service Discoverability and Service Models 378
 - How Service Discoverability Affects Other Principles 378

- 12.6 Risks Associated with Service Discoverability 381
 - Post-Implementation Application of Discoverability 381
 - Application of this Principle by Non-Communicative Resources 381
- 12.7 Case Study Example 382
 - Service Profiles (Functional Meta Information) 382
 - Related Quality of Service Meta Information 386

Chapter 13: Service Composability (Composition Member Design and Complex Compositions) 387

- 13.1 Composition Explained 388
 - Composition in Abstract 388
 - Origins of Composition 390
- 13.2 Profiling this Principle 392
- 13.3 Composition Concepts and Terminology 396
 - Compositions and Composition Instances 397
 - Composition Members and Controllers 398
 - Service Compositions and Web Services 401
 - Service Activities 402
 - Composition Initiators 403
 - Point-to-Point Data Exchanges and Compositions 405
 - Types of Compositions 406
- 13.4 The Complex Service Composition 407
 - Stages in the Evolution of a Service Inventory 407
 - Defining the Complex Service Composition 410
 - Preparing for the Complex Service Composition 411
- 13.5 Measuring Service Composability and Composition
 - Effectiveness Potential 412
 - Evolutionary Cycle States of a Composition 412
 - Composition Design Assessment 413
 - Composition Runtime Assessment 415
 - Composition Governance Assessment 417
 - Measuring Composability 419

- 13.6 Composition and Service Design 427
 - Service Composability and Granularity 427
 - Service Composability and Service Models 428
 - Service Composability and Composition Autonomy 430
 - Service Composability and Orchestration 430
 - How Service Composability Affects Other Principles 432
- 13.7 Risks Associated with Service Composition 437
 - Composition Members as Single Points of Failure 437
 - Composition Members as Performance Bottlenecks 437
 - Governance Rigidity of “Over-Reuse” in Compositions 438
- 13.8 Case Study Example 439

PART III: SUPPLEMENTAL

Chapter 14: Service-Orientation and Object-Oriented: A Comparison of Principles and Concepts 445

- 14.1 A Tale of Two Design Paradigms 446
- 14.2 A Comparison of Goals 449
 - Increased Business Requirements Fulfillment 450
 - Increased Robustness 451
 - Increased Extensibility 451
 - Increased Flexibility 452
 - Increased Reusability and Productivity 452
- 14.3 A Comparison of Fundamental Concepts 453
 - Classes and Objects 453
 - Methods and Attributes 454
 - Messages 454
 - Interfaces 456
- 14.4 A Comparison of Design Principles 457
 - Encapsulation 458
 - Inheritance 459

- Generalization and Specialization 461
- Abstraction 463
- Polymorphism 463
- Open-Closed Principle (OCP) 465
- Don't Repeat Yourself (DRY) 465
- Single Responsibility Principle (SRP) 466
- Delegation 468
- Association 469
- Composition 470
- Aggregation 471
- 14.5 Guidelines for Designing Service-Oriented Classes. 472
 - Implement Class Interfaces 473
 - Limit Class Access to Interfaces 473
 - Do Not Define Public Attributes in Interfaces 473
 - Use Inheritance with Care 473
 - Avoid Cross-Service “has-a” Relationships 474
 - Use Abstract Classes for Modeling, Not Design 474
 - Use Façade Classes 474

Chapter 15: Supporting Practices 477

- 15.1 Service Profiles 478
 - Service-Level Profile Structure 478
 - Capability Profile Structure 480
 - Additional Considerations 482
- 15.2 Vocabularies 483
 - Service-Oriented Computing Terms 484
 - Service Classification Terms 484
 - Types and Associated Terms 485
 - Design Principle Application Levels 487
- 15.3 Organizational Roles 488
 - Service Analyst 490
 - Service Architect 490
 - Service Custodian 491
 - Schema Custodian 491
 - Policy Custodian 492

Service Registry Custodian 492
 Technical Communications Specialist 493
 Enterprise Architect 493
 Enterprise Design Standards Custodian (and Auditor) 494

Chapter 16: Mapping Service-Orientation Principles to Strategic Goals 497

16.1 Principles that Increase Intrinsic Interoperability 498
 16.2 Principles that Increase Federation 501
 16.3 Principles that Increase Vendor Diversification Options . 501
 16.4 Principles that Increase Business and Technology Domain Alignment 502
 16.5 Principles that Increase ROI 504
 16.6 Principles that Increase Organizational Agility 505
 16.7 Principles that Reduce the Overall Burden of IT 507

PART IV: APPENDICES

Appendix A: Case Study Conclusion 513

Appendix B: Process Descriptions 517

B.1 Delivery Processes 518
 Bottom-Up vs. Top-Down 518
 The Inventory Analysis Cycle 520
 Inventory Analysis and Service-Oriented Design 521
 Choosing a Delivery Strategy 521
 B.2 Service-Oriented Analysis Process 522
 Define Analysis Scope 522
 Identify Affected Systems 523
 Perform Service Modeling 523

B.3 Service Modeling Process 523

B.4 Service-Oriented Design Processes. 525

 Design Processes and Service Models 526

 Service Design Processes and Service-Orientation 527

Appendix C: Principles and Patterns

Cross-Reference 529

Additional Resources 533

About the Author 535

About the Photos 537

Index 539

Preface

Over the past few years I've been exposed to many different IT environments as part of a wide range of SOA initiatives for clients in both private and public sectors. While doing some work on a project for a client in the defense industry, I had an opportunity to learn more about not just their technical landscape, but also the various policies and procedures that are specific to the defense culture. During this time I came across the DoD Standardization Program, an initiative comprised of documents and specifications that establish guiding principles and standards for various aspects of the military, including the design of weapons and military equipment, as well as the definition of methods and processes used by military personnel.

While reading about this program, I learned that several other standardization programs have been in existence for some time, facilitating standardization within public sector organizations (such as the Coast Guard and NASA), as well as numerous private sector industries. The goals of these programs tend to revolve around the establishment of industry standards to enhance interoperability with the ultimate objective of reducing operational overhead, reducing risk, and increasing the organization's overall effectiveness.

In the case of the aforementioned public sector-related standards, interoperability may refer to the exchange of equipment or weapons or the exchange and collaboration of personnel from different locations.

For example, an ammunition clip manufactured in Iowa, stored in Virginia, and delivered to and used by someone at a training base in Texas will work perfectly with a gun manufactured in Kansas because both of these products were built according to the same set of specifications. Similarly, in response to a natural disaster a rescue team may

need to be quickly assembled from individuals based out of different cities and who have never previously worked together. This team can still function effectively because all team members were trained as per the same procedures and processes, using the same vocabulary and conventions.

These standardization programs have much in common with the rationale and objectives behind SOA and service-orientation. The fundamental goal is to produce something with repeatable value, long-term benefit, and inherent flexibility, all for the strategic good of the organization. The greatest obstacle to achieving this goal in the world of SOA has been a lack of understanding as to what service-orientation, as an industry paradigm, really is. It is my hope that this book will help rectify this situation by providing some clarity for what it means for something to be “service-oriented.”

Acknowledgments

To ensure the accuracy and legitimacy of the content in this book, I decided early on to subject it to a rigorous quality assurance process that involved technical reviews by over 60 industry professionals. I am deeply grateful for the time and effort these individuals dedicated to these reviews. Specifically, I would like to thank Kevin Davis, PhD, Ronald Bourret, Robert Schneider, Ravi Palepu, Wes McGregor, Judith Myerson, and Cyrille Thilloy for their early feedback, and the following technical reviewers that participated in the full manuscript review (in alphabetical order by last name):

Dr. Mohamad Afshar, Oracle Corporation

Wayne Ariola, Parasoft

Raj Balasubramanian, IBM Software Group

Stephen Bennett, BEA Systems, Inc.

Steve Birkel, Intel Corporation

Brandon Bohling, Intel Corporation

Peter Chang, PhD, Lawrence Technological University

Robin Chen, PhD, Google, Inc.

Jim Clune, Parasoft

Jason “AJ” Comfort Sr., Booz Allen Hamilton, Inc.

Bill Draven, Intel Corporation

Darryl Hogan, Microsoft Corporation

Continues

Fred Ingham, Platinum Solutions Inc.
Cory Isaacson, Rogue Wave Software
Radovan Janecek, Hewlett-Packard
Anish Karmarkar, Oracle Corporation
Hanu Kommalapati, Microsoft Corporation
Robert Laird, IBM EAI/SOA Advanced Technologies Group
Dr. Mark Little, Redhat
Canyang Kevin Liu, SAP Americas, Inc.
David Michalowicz, MITRE Corporation
Jim Murphy, Mindreef, Inc.
Prakash Narayan, Sun Microsystems
Philipp Offermann, University of Berlin
James Pasley, Cape Clear Software
Robin G. Qiu, PhD, Pennsylvania State University
Christoph Schittko, Microsoft Corporation
Dr. Arnaud Simon, Redhat
R. Perry Smith, EDS/OnStar
Michael H. Sor, Booz Allen Hamilton, Inc.
Philip Thomas, IBM United Kingdom Limited
Andre Tost, IBM Software Group
Sameer Tyagi, Fidelity Investments
Umit Yalcinalp, SAP
Farzin Yashar, IBM SOA Advanced Technologies
Kareem Yusuf, IBM Software Group
Markus Zirn, Oracle Corporation

Chapter 4



Service-Orientation

- 4.1 Introduction to Service-Orientation
- 4.2 Problems Solved by Service-Orientation
- 4.3 Challenges Introduced by Service-Orientation
- 4.4 Additional Considerations
- 4.5 Effects of Service-Orientation on the Enterprise
- 4.6 Origins and Influences of Service-Orientation
- 4.7 Case Study Background

Having covered some of the basic elements of service-oriented computing, we now narrow our focus on service-orientation. The next set of sections establish the paradigm of service-orientation and explain how it is changing the face of distributed computing.

4.1 Introduction to Service-Orientation

In the every day world around us, services are and have been commonplace for as long as civilized history has existed. Any person carrying out a distinct task in support of others is providing a service (Figure 4.1). Any group of individuals collectively performing a task is also demonstrating the delivery of a service.

Figure 4.1
Three individuals, each capable of providing a distinct service.

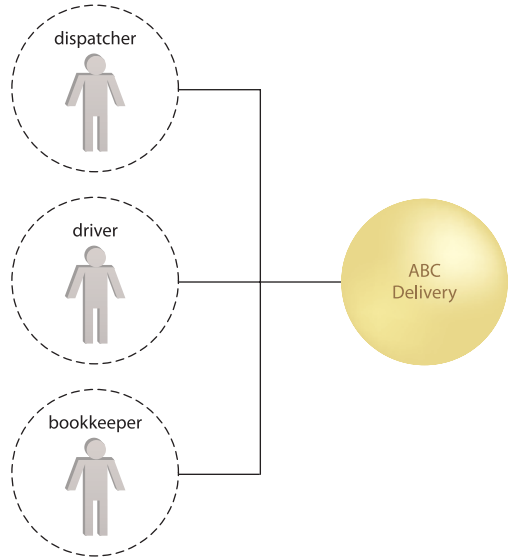


Similarly, an organization that carries out tasks associated with its purpose or business is also providing a service. As long as the task or function being provided is well-defined and can be relatively isolated from other associated tasks, it can be distinctly classified as a service (Figure 4.2).

Certain baseline requirements exist to enable a group of individual service providers to collaborate in order to collectively provide a larger service. Figure 4.2, for example, displays a group of employees that each provide a service for ABC Delivery. Even though each individual contributes a distinct service, for the company to function effectively, its staff also needs to have fundamental, common characteristics, such as availability, reliability, and the ability to communicate using the same language. With all of this in place, these individuals can be composed into a productive working team. Establishing these types of baseline requirements is a key goal of service-orientation.

Figure 4.2

A company that employs these three people can compose their capabilities to carry out its business.



Services in Business Automation

In the world of SOA and service-orientation, the term “service” is not generic. It has specific connotations that relate to a unique combination of design characteristics. When solution logic is consistently built as services and when services are consistently designed with these common characteristics, service-orientation is successfully realized throughout an environment.

For example, one of the primary service design characteristics explored as part of this study of service-orientation is reusability. A strong emphasis on producing solution logic in the format of services that are positioned as highly generic and reusable enterprise resources gradually transitions an organization to a state where more and more of its solution logic becomes less dependent on and more agnostic to any one purpose or business process. Repeatedly fostering this characteristic within services eventually results in wide-spread reuse potential.

Consistently realizing specific design characteristics requires a set of guiding principles. This is what the service-orientation design paradigm is all about.

Services Are Collections of Capabilities

When discussing services, it is important to remember that a single service can provide a collection of capabilities. They are grouped together because they relate to a functional

context established by the service. The functional context of the service illustrated in Figure 4.3, for example, is that of “shipment.” Therefore, this particular service provides a set of capabilities associated with the processing of shipments.

Figure 4.3

Much like a human, an automated service can provide multiple capabilities.



A service can essentially act as a container of related capabilities. It is comprised of a body of logic designed to carry out these capabilities and a service contract that expresses which of its capabilities are made available for public invocation.

References to service capabilities in this book are specifically focused on those that are defined in the service contract. For a discussion of how service capabilities are distinguished from Web service operations and component methods, see the *Principles and Service Implementation Mediums* section in Chapter 5.

Service-Orientation as a Design Paradigm

As established in Chapter 3, a design paradigm is an approach to designing solution logic. When building distributed solution logic, design approaches revolve around a software engineering theory known as the *separation of concerns*. In a nutshell, this theory states that a larger problem is more effectively solved when decomposed into a set of smaller problems or *concerns*. This gives us the option of partitioning solution logic into capabilities, each designed to solve an individual concern. Related capabilities can be grouped into units of solution logic.

The fundamental benefit to solving problems this way is that a number of the solution logic units can be designed to solve immediate concerns while still remaining agnostic to the greater problem. This provides the constant opportunity for us to reutilize the capabilities within those units to solve other problems as well.

Different design paradigms exist for distributed solution logic. What distinguishes service-orientation is the manner in which it carries out the separation of concerns and how it shapes the individual units of solution logic. Applying service-orientation to a meaningful extent results in solution logic that can be safely classified as “service-oriented”

and units that qualify as “services.” To understand exactly what that means requires an appreciation of the strategic goals covered in Chapter 3 combined with knowledge of the associated design principles documented in Part II.

For now, let’s briefly introduce each of these principles:

Standardized Service Contract

Services express their purpose and capabilities via a service contract. The Standardized Service Contract design principle is perhaps the most fundamental part of service-orientation in that it essentially requires that specific considerations be taken into account when designing a service’s public technical interface and assessing the nature and quantity of content that will be published as part of a service’s official contract.

A great deal of emphasis is placed on specific aspects of contract design, including the manner in which services express functionality, how data types and data models are defined, and how policies are asserted and attached. There is a constant focus on ensuring that service contracts are both optimized, appropriately granular, and standardized to ensure that the endpoints established by services are consistent, reliable, and governable.

Chapter 6 is dedicated to exploring this design principle in detail.

Service Loose Coupling

Coupling refers to a connection or relationship between two things. A measure of coupling is comparable to a level of dependency. This principle advocates the creation of a specific type of relationship within and outside of service boundaries, with a constant emphasis on reducing (“loosening”) dependencies between the service contract, its implementation, and its service consumers.

The principle of Service Loose Coupling promotes the independent design and evolution of a service’s logic and implementation while still guaranteeing baseline interoperability with consumers that have come to rely on the service’s capabilities. There are numerous types of coupling involved in the design of a service, each of which can impact the content and granularity of its contract. Achieving the appropriate level of coupling requires that practical considerations be balanced against various service design preferences.

Chapter 7 provides an in-depth exploration of this principle and introduces related patterns and concepts.

Service Abstraction

Abstraction ties into many aspects of service-orientation. On a fundamental level, this principle emphasizes the need to hide as much of the underlying details of a service as possible. Doing so directly enables and preserves the previously described loosely coupled relationship. Service Abstraction also plays a significant role in the positioning and design of service compositions.

Various forms of meta data come into the picture when assessing appropriate abstraction levels. The extent of abstraction applied can affect service contract granularity and can further influence the ultimate cost and effort of governing the service.

Chapter 8 covers several aspects of applying abstraction to different types of service meta data, along with processes and approaches associated with information hiding.

Service Reusability

Reuse is strongly advocated within service-orientation; so much so, that it becomes a core part of typical service analysis and design processes, and also forms the basis for key service models. The advent of mature, non-proprietary service technology has provided the opportunity to maximize the reuse potential of multi-purpose logic on an unprecedented level.

The principle of Service Reusability emphasizes the positioning of services as enterprise resources with agnostic functional contexts. Numerous design considerations are raised to ensure that individual service capabilities are appropriately defined in relation to an agnostic service context, and to guarantee that they can facilitate the necessary reuse requirements.

Variations and levels of reuse and associated agnostic service models are covered in Chapter 9, along with a study of how commercial product design approaches have influenced this principle.

Service Autonomy

For services to carry out their capabilities consistently and reliably, their underlying solution logic needs to have a significant degree of control over its environment and resources. The principle of Service Autonomy supports the extent to which other design principles can be effectively realized in real world production environments by fostering design characteristics that increase a service's reliability and behavioral predictability.

This principle raises various issues that pertain to the design of service logic as well as the service's actual implementation environment. Isolation levels and service normalization considerations are taken into account to achieve a suitable measure of autonomy, especially for reusable services that are frequently shared.

Chapter 10 documents the design issues and challenges related to attaining higher levels of service autonomy, and further classifies different forms of autonomy and highlights associated risks.

Service Statelessness

The management of excessive state information can compromise the availability of a service and undermine its scalability potential. Services are therefore ideally designed to remain stateful only when required. Applying the principle of Service Statelessness requires that measures of realistically attainable statelessness be assessed, based on the adequacy of the surrounding technology architecture to provide state management delegation and deferral options.

Chapter 11 explores the options and impacts of incorporating stateless design characteristics into service architectures.

Service Discoverability

For services to be positioned as IT assets with repeatable ROI they need to be easily identified and understood when opportunities for reuse present themselves. The service design therefore needs to take the “communications quality” of the service and its individual capabilities into account, regardless of whether a discovery mechanism (such as a service registry) is an immediate part of the environment.

The application of this principle, as well as an explanation of how discoverability relates to interpretability and the overall service discovery process, are covered in Chapter 12.

Service Composability

As the sophistication of service-oriented solutions continues to grow, so does the complexity of underlying service composition configurations. The ability to effectively compose services is a critical requirement for achieving some of the most fundamental goals of service-oriented computing.

Complex service compositions place demands on service design that need to be anticipated to avoid massive retro-fitting efforts. Services are expected to be capable of participating as effective composition members, regardless of whether they need to be immediately enlisted in a composition. The principle of Service Composability addresses this requirement by ensuring that a variety of considerations are taken into account.

How the application of this design principle helps prepare services for the world of complex compositions is described in Chapter 13.

Service-Orientation and Interoperability

One item that may appear to be absent from the preceding list is a principle along the lines of “*Services are Interoperable.*” The reason this does not exist as a separate principle is because interoperability is fundamental to every one of the principles just described. Therefore, in relation to service-oriented computing, stating that services must be interoperable is just about as basic as stating that services must exist. Each of the eight principles supports or contributes to interoperability in some manner.

Here are just a few examples:

- Service contracts are standardized to guarantee a baseline measure of interoperability associated with the harmonization of data models.
- Reducing the degree of service coupling fosters interoperability by making individual services less dependent on others and therefore more open for invocation by different service consumers.
- Abstracting details about the service limits all interoperation to the service contract, increasing the long-term consistency of interoperability by allowing underlying service logic to evolve more independently.
- Designing services for reuse implies a high-level of required interoperability between the service and numerous potential service consumers.
- By raising a service’s individual autonomy, its behavior becomes more consistently predictable, increasing its reuse potential and thereby its attainable level of interoperability.
- Through an emphasis on stateless design, the availability and scalability of services increase, allowing them to interoperate more frequently and reliably.

- Service Discoverability simply allows services to be more easily located by those who want to potentially interoperate with them.
- Finally, for services to be effectively composable they must be interoperable. The success of fulfilling composability requirements is often tied directly to the extent to which services are standardized and cross-service data exchange is optimized.

A fundamental goal of applying service-orientation is for interoperability to become a natural by-product, ideally to the extent that a level of intrinsic interoperability is established as a common and expected service design characteristic. Depending on the architectural strategy being employed, this extent may or may not be limited to a specific service inventory.

Of course, as with any other design characteristic, there are levels of interoperability a service can attain. The ultimate measure is generally determined by the extent to which service-orientation principles have been consistently and successfully realized (plus, of course, environmental factors such as the compatibility of wire protocols, the maturity level of the underlying technology platform, and adherence to technology standards).

NOTE

Increased intrinsic interoperability is one of the key strategic goals associated with service-oriented computing (as originally established in Chapter 3). For more detailed information about how service-orientation principles directly support this and other strategic goals, see Chapter 16.

SUMMARY OF KEY POINTS

- The service-orientation paradigm consists of eight distinct design principles, each of which fosters fundamental design characteristics, such as interoperability. These principles are explored individually in subsequent chapters.
- Interoperability is a natural by-product of applying service-orientation design principles.

4.2 Problems Solved by Service-Orientation

To best appreciate why service-orientation has emerged and how it is intended to improve the design of automation systems, we need to compare before and after perspectives. By studying some of the common issues that have historically plagued IT, we can begin to understand the solutions proposed by this design paradigm.

NOTE

This book fully acknowledges that past design paradigms have advocated similar principles and strategic goals as service-orientation. Several of these design approaches, in fact, directly inspired or influenced service-orientation (as explained further in the *Origins and Influences of Service-Oriented Architecture* section of this chapter). The following section is focused specifically on a comparison with the silo-based design approach because it has persisted as the most common means by which applications are delivered.

Life Before Service-Oriented Architecture

In the world of business it makes a great deal of sense to deliver solutions capable of automating the execution of business tasks. Over the course of IT's history, the majority of such solutions have been created with a common approach of identifying the business tasks to be automated, defining their business requirements, and then building the corresponding solution logic (Figure 4.4).

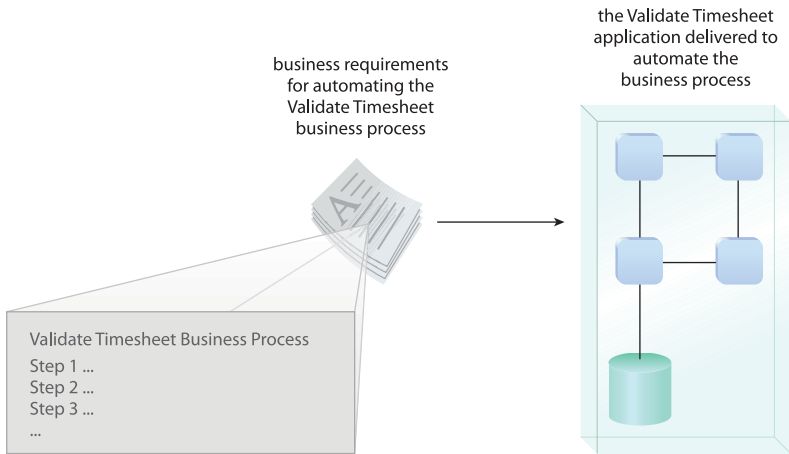


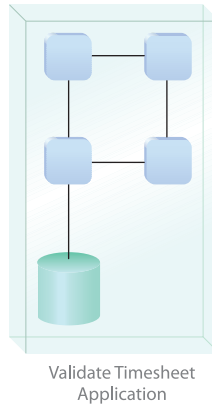
Figure 4.4

A ratio of one application for each new set of automation requirements has been common.

This has been an accepted and proven approach to achieving tangible business benefits through the use of technology and has been successful at providing a relatively predictable return on investment (Figure 4.5).

Figure 4.5

A sample formula for calculating ROI is based on a predetermined investment with a predictable return.



Development cost = x
 Yearly operational cost = y
 Estimated yearly savings
 due to increased productivity = $(x/2) - y$

The ability to gain any further value from these applications is usually inhibited because their capabilities are tied to specific business requirements and processes (some of which will even have a limited lifespan). When new requirements and processes come our way, we are forced to either make significant changes to what we already have, or we may need to build a new application altogether.

In the latter case, although repeatedly building “disposable applications” is not the perfect approach, it has proven itself as a legitimate means of automating business. Let’s explore some of the lessons learned by first focusing on the positive.

- Solutions can be built efficiently because they only need to be concerned with the fulfillment of a narrow set of requirements associated with a limited set of business processes.
- The business analysis effort involved with defining the process to be automated is straight forward. Analysts are focused only on one process at a time and therefore only concern themselves with the business entities and domains associated with that one process.
- Solution designs are tactically focused. Although complex and sophisticated automation solutions are sometimes required, the sole purpose of each is to automate just one or a specific set of business processes. This predefined functional scope simplifies the overall solution design as well as the underlying application architecture.

- The project delivery lifecycle for each solution is streamlined and relatively predictable. Although IT projects are notorious for being complex endeavors, riddled with unforeseen challenges, when the delivery scope is well-defined (and doesn't change), the process and execution of the delivery phases have a good chance of being carried out as expected.
- Building new systems from the ground up allows organizations to take advantage of the latest technology advancements. The IT marketplace progresses every year to the extent that we fully expect technology we use to build solution logic today to be different and better tomorrow. As a result, organizations that repeatedly build disposable applications can leverage the latest technology innovations with each new project.

These and other common characteristics of traditional solution delivery provide a good indication as to why this approach has been so popular. Despite its acceptance, though, it has become evident that there is still lots of room for improvement.

It Can Be Highly Wasteful

The creation of new solution logic in a given enterprise commonly results in a significant amount of redundant functionality (Figure 4.6). The effort and expense required to construct this logic is therefore also redundant.

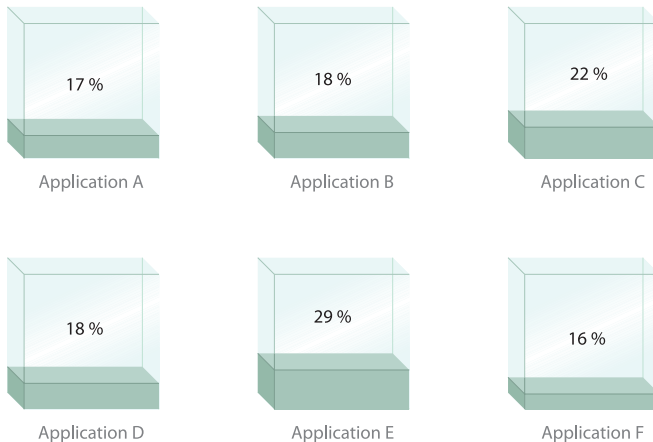


Figure 4.6

Different applications developed independently can result in significant amounts of redundant functionality. The applications displayed were delivered with various levels of solution logic that, in some form, already existed.

It's Not as Efficient as it Appears

Because of the tactical focus on delivering solutions for specific process requirements, the scope of development projects is highly targeted. Therefore, there is the constant perception that business requirements will be fulfilled at the earliest possible time. However, by continually building and rebuilding logic that already exists elsewhere, the process is not as efficient as it could be if the creation of redundant logic could be avoided (Figure 4.7).

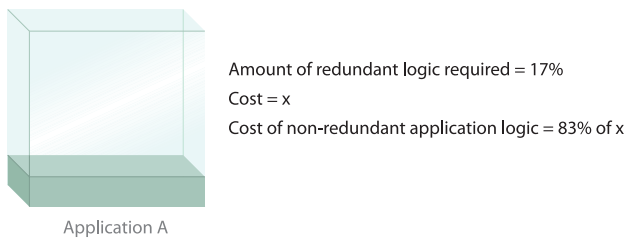


Figure 4.7
Application A was delivered for a specific set of business requirements. Because a subset of these business requirements had already been fulfilled elsewhere, Application A's delivery scope is larger than it has to be.

It Bloats an Enterprise

Each new or extended application adds to the bulk of an IT environment's system inventory (Figure 4.8). The ever-expanding hosting, maintenance, and administration demands can inflate an IT department in budget, resources, and size to the extent that IT becomes a significant drain on the overall organization.

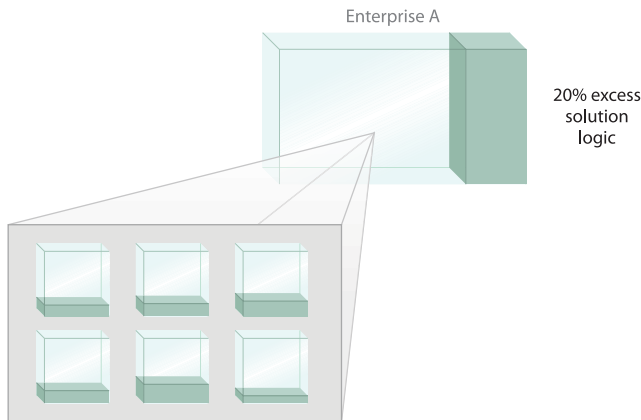


Figure 4.8
This simple diagram portrays an enterprise environment containing applications with redundant functionality. The net effect is a larger enterprise.

It Can Result in Complex Infrastructures and Convoluted Enterprise Architectures

Having to host numerous applications built from different generations of technologies and perhaps even different technology platforms often requires that each will impose unique architectural requirements. The disparity across these “siloed” applications can lead to a counter-federated environment (Figure 4.9), making it challenging to plan the evolution of an enterprise and scale its infrastructure in response to that evolution.

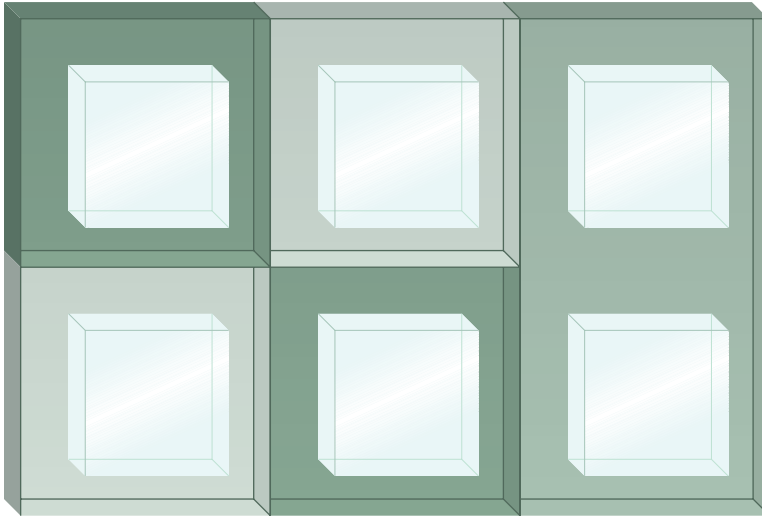


Figure 4.9

Different application environments within the same enterprise can introduce incompatible runtime platforms as indicated by the shaded zones.

Integration Becomes a Constant Challenge

Applications built only with the automation of specific business processes in mind are generally not designed to accommodate other interoperability requirements. Making these types of applications share data at some later point results in a jungle of convoluted integration architectures held together mostly through point-to-point patchwork (Figure 4.10) or requiring the introduction of large middleware layers.

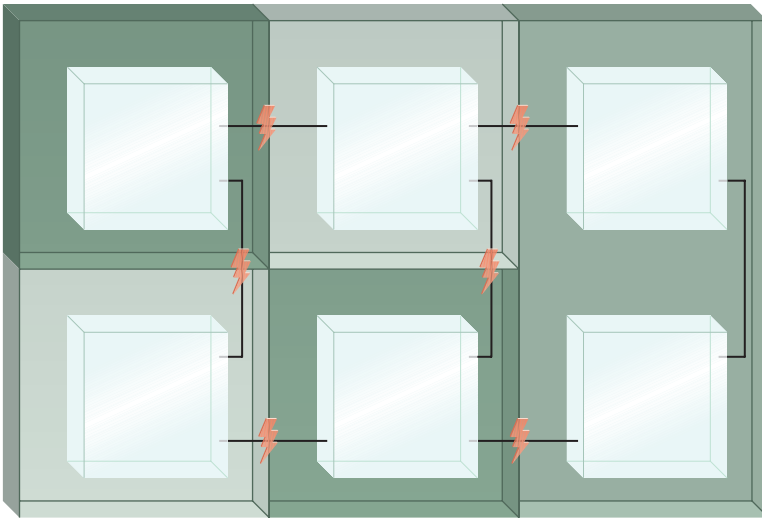


Figure 4.10

A vendor-diverse enterprise can introduce a variety of integration challenges, as expressed by the little lightning bolts that highlight points of concern when trying to bridge proprietary environments.

The Need for Service-Orientation

After repeated generations of traditional distributed solutions, the severity of the previously described problems has been amplified. This is why service-orientation was conceived. It very much represents an evolutionary state in the history of IT in that it combines successful design elements of past approaches with new design elements that leverage conceptual and technology innovation.

The consistent application of the eight design principles listed earlier results in the widespread proliferation of the corresponding design characteristics:

- increased consistency in how functionality and data is represented
- reduced dependencies between units of solution logic
- reduced awareness of underlying solution logic design and implementation details
- increased opportunities to use a piece of solution logic for multiple purposes
- increased opportunities to combine units of solution logic into different configurations

- increased behavioral predictability
- increased availability and scalability
- increased awareness of available solution logic

When these characteristics exist as real parts of implemented services, they establish a common synergy. As a result, the complexion of an enterprise changes as the following distinct qualities are consistently promoted:

Increased Amounts of Agnostic Solution Logic

Within a service-oriented solution, units of logic (services) encapsulate functionality not specific to any one application or business process (Figure 4.11). These services are therefore classified as agnostic and reusable IT assets.

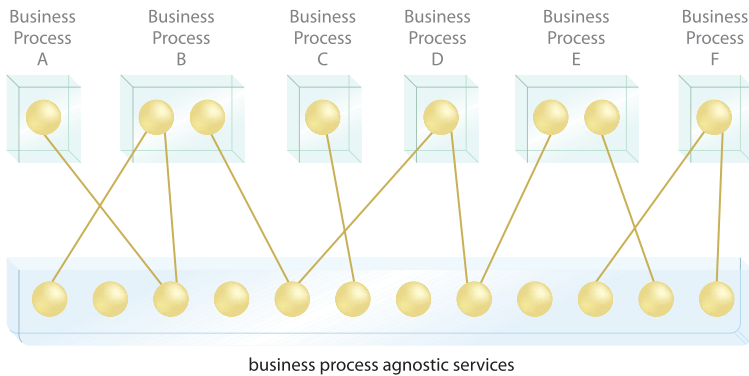


Figure 4.11
Business processes are automated by a series of business process-specific services (top layer) that share a pool of business process-agnostic services (bottom layer). These layers correspond to the task, entity, and utility service models described in Chapter 3.

Reduced Amounts of Application-Specific Logic

Increasing the amount of solution logic not specific to any one application or business process decreases the amount of required application-specific logic (Figure 4.12). This blurs the lines between standalone application environments by reducing the overall quantity of standalone applications. (See also the *Service-Orientation and the Concept of “Application”* section later in this chapter.)

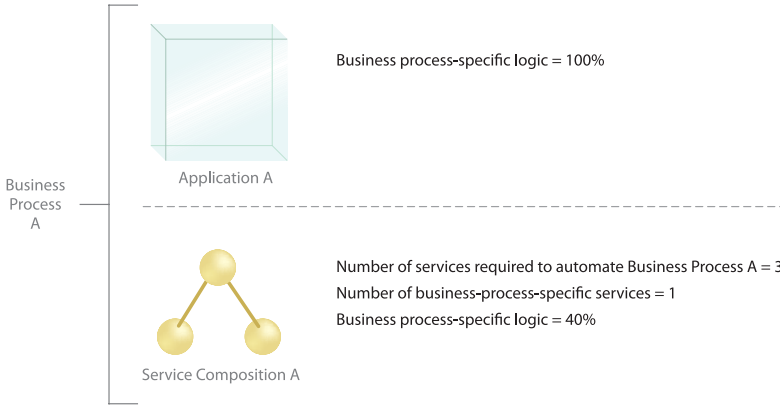


Figure 4.12

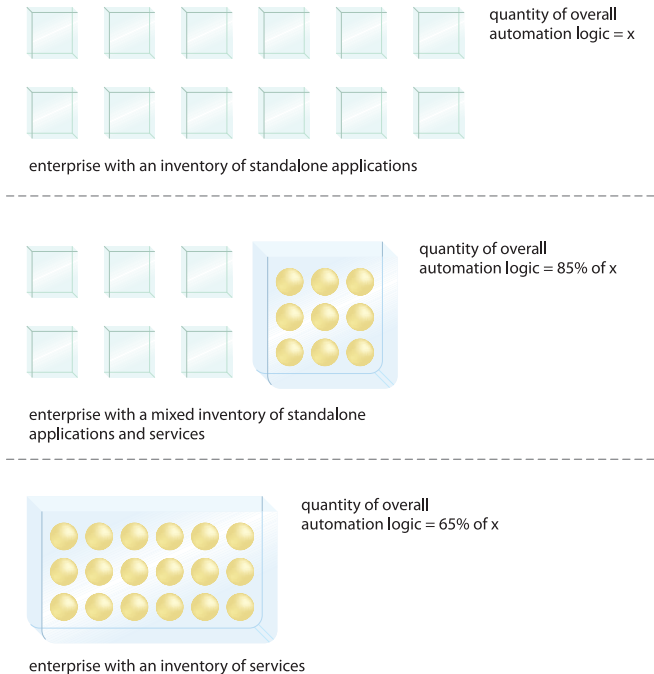
Business Process A can be automated by either Application A or Service Composition A. The delivery of Application A can result in a body of solution logic that is specific to and tailored for the business process. Service Composition A would be designed to automate the process with a combination of agnostic services and 40% of additional logic specific to the business process.

Reduced Volume of Logic Overall

The overall quantity of solution logic is reduced because the same solution logic is shared and reused to automate multiple business processes, as shown in Figure 4.13.

Figure 4.13

The quantity of solution logic shrinks as an enterprise transitions toward a standardized service inventory comprised of “normalized” services.



Inherent Interoperability

Common design characteristics consistently implemented result in solution logic that is naturally aligned. When this carries over to the standardization of service contracts and their underlying data models, a base level of automatic interoperability is achieved across services, as illustrated in Figure 4.14. (See also the *Service-Oriented and the Concept of “Integration”* section later in this chapter.)

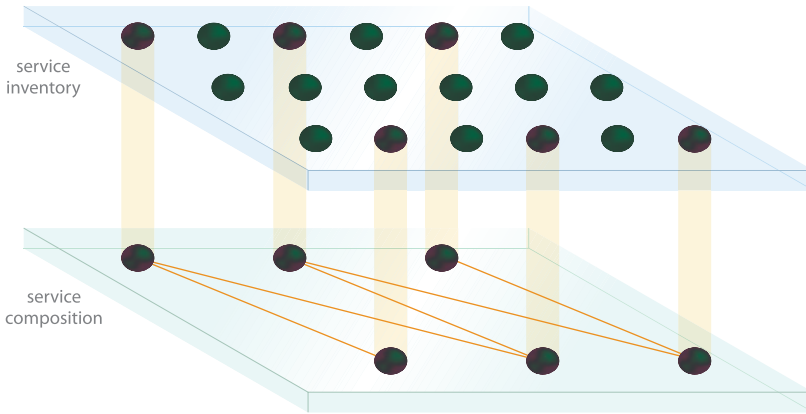


Figure 4.14

Services from different parts of a service inventory can be combined into new compositions. If these services are designed to be intrinsically interoperable, the effort to assemble them into new composition configurations is significantly reduced.

SUMMARY OF KEY POINTS

- The traditional silo-based approach to building applications has been successful at providing tangible benefits and measurable returns on investment.
 - This approach has also caused its share of problems, most notably an increase in integration complexity and an increase in the size and administrative burden of IT enterprises.
 - Service-orientation establishes a design paradigm that leverages and builds upon previous approaches and proposes a means of avoiding problems associated with silo-based application delivery.
-

4.3 Challenges Introduced by Service-Orientation

As much as service-orientation can solve some of the more significant historical problems in IT, its application in the real world can make some serious impositions. It is necessary to be aware of these challenges ahead of time because being prepared is key to overcoming them.

Design Complexity

With a constant emphasis on reuse, a significant percentage of a service inventory can ultimately be comprised of agnostic services capable of fulfilling requirements for multiple potential service consumer programs.

Although this can establish a highly normalized and streamlined architecture, it can also introduce an increased level of complexity for both the architecture as well as individual service designs.

Examples include:

- increased performance requirements resulting from the increased reuse of agnostic services
- reliability issues of services at peak concurrent usage times and availability issues of services during off-hours
- single point of failure issues introduced by excessive reuse of agnostic services (and that may require the need for redundant deployments to mitigate risks)
- increased demands on service hosting environments to accommodate autonomy-related preferences
- service contract versioning issues and the impact of potentially redundant service contracts

Design issues such as these can be addressed by a combination of sound technology architecture design, modern vendor runtime platform technology, and the consistent application of service-orientation design principles. Solving service reliability and performance issues in particular are primary goals of those design principles more focused on the underlying service logic, such as Service Autonomy, Service Statelessness, and Service Composability.

The Need for Design Standards

Design standards can be healthy for an enterprise in that they “pre-solve” problems by making several decisions for architects and developers ahead of time, thereby increasing the consistency and compatibility of solution designs. Their use is required in order to realize the successful propagation of service-orientation.

Although it can be a straight-forward process to create these standards, incorporating them into a (non-standardized) IT culture already set in its ways can be demanding to say the least. The usage of design standards can introduce the need to enforce their compliance, a policing role that can meet with resistance. Additionally, architects and developers sometimes feel that design standards inhibit their creativity and ability to innovate.

A circumstance that tends to aid the large-scale realization of standardization is when the SOA initiative is championed by an executive manager, such as a CIO. When an individual or a governing body has the authority to essentially “lay down the law,” many of these cultural issues resolve themselves more quickly. However, within organizations based on peer-level departmental structures (which are more common in the public sector), the acceptance of design standards may require negotiation and compromise.

The best weapon for overcoming cultural resistance to design standards is communication and education. Those resisting standardization efforts are more likely to become supporters after gaining an appreciation of the strategic significance and ultimate benefits of adopting and respecting the need for design standards.

Top-Down Requirements

A preferred strategy to delivering services is to first conceptualize a service inventory by defining a blueprint of all planned services, their relationships, boundaries, and individual service models. This approach is very much associated with a top-down delivery strategy in that it can impose a significant amount of up-front analysis effort involving many members of business analysis and technology architecture groups.

Though preferred, achieving a comprehensive blueprint prior to building services is often not feasible. It is common for organizations to face budget and time constraints and tactical priorities that simply won’t permit it. As a result, there are phased and iterative delivery approaches that allow for services to be produced earlier on. These, however, often come with trade-offs in that they can require the service designs to be revisited and revised at a later point. While this can introduce risks associated with

the implementation of premature service designs, it is often considered an acceptable compromise.

The principles of service-orientation can be applied to services on an individual basis, allowing a reasonable degree of service-orientation to be achieved regardless of the approach. However, the actual quality of the resulting service designs is typically tied to how much of the top-down analysis work was completed prior to their delivery.

BEST PRACTICE

It is recommended that, at minimum, a high-level service inventory blueprint always be defined prior to creating physical service contracts. This establishes an important “broader” perspective in support of service-oriented analysis and service modeling processes and, ultimately, results in stronger and more durable service designs.

Counter-Agile Service Delivery in Support of Agile Solution Delivery

Irrespective of the potential top-down efforts needed for some SOA projects, the additional design considerations required to implement a meaningful measure of each of the eight design principles increases both the overall time and cost to deliver service logic.

This may appear contrary to the attention SOA has received for its ability to increase agility. To achieve the state of organizational agility described in Chapter 3 requires that service-orientation already be successfully implemented. This is what establishes an environment in which the delivery of solutions is much more agile.

However, given that it takes more initial effort to design and build services than it does to build a corresponding amount of logic that is not service-oriented, the process of delivering services in support of SOA can actually be *counter-agile*. This can cause issues for an organization that has tactical requirements or needs to be responsive while building a service inventory.

BEST PRACTICE

An effective approach, when sufficient resources are available, is to allow SOA initiatives to be delivered alongside existing legacy development and maintenance projects. This way, tactical requirements can continue to be fulfilled by traditional applications while the enterprise works toward a phased transition toward service-oriented computing.

Appendix B provides additional coverage of SOA delivery strategies that address tactical versus strategic service delivery requirements.

Governance Demands

The eventual existence of one or more service inventories represents the ultimate deliverable of the typical large-scale SOA initiative. A service inventory establishes a powerful reserve of standardized solution logic, a high percentage of which will ideally be classified as agnostic or reusable. Subsequent to their implementation, though, the management and evolution of these agnostic services can be responsible for some of the most profound changes imposed by service-orientation.

In the past, a standalone application was typically developed by a single project team. Members of this team often ended up remaining “attached” to the application for subsequent upgrades, maintenance, and extensions. This ownership model worked because the application’s overall purpose and scope remained focused on the business tasks it was originally built to automate.

The body of solution logic represented by agnostic services, however, is intentionally positioned to *not* belong to any one business process. Although these services may have been delivered by a project team, that same team may not continue to own the service logic as it gets repeatedly utilized by other solutions, processes, and compositions.

Therefore, a special governance structure is required. This can introduce new resources, roles, processes, and even new groups or departments. Ultimately, when these issues are under control and the IT environment itself has successfully adapted to the required changes, the many benefits associated with this new computing platform are there for the taking. However, the process of moving to this new governance model can challenge traditional approaches and demand time, expense, and a great deal of patience.

SUMMARY OF KEY POINTS

- Applying service-orientation on a broad scale can introduce increased design complexity and the need for a consistent level of standardization.
 - The construction of services can be expensive and time-consuming, introducing a more burdensome project delivery lifecycle, further compounded by some of the common top-down analysis requirements that may need to be in place before services can be built.
 - Service inventory governance requirements can impose significant changes that can shake up the organizational structure of an IT department.
-

4.4 Additional Considerations

To supplement the benefits and challenges just covered, this section discusses some further aspects of service-orientation.

It Is Not a Revolutionary Paradigm

Service-orientation is not a brand new paradigm that aims to replace all that preceded it. It, in fact, incorporates and builds upon proven and successful elements from past paradigms and combines these with design approaches shaped to leverage recent technology innovations.

This is why we do not refer to SOA as a revolutionary model in the history of IT. It is simply the next stage in an evolutionary cycle that began with the application of modularity on a small scale (by organizing simple programming routines into shared modules for example) and has now spread to the potential modularization of the enterprise.

Enterprise-wide Standardization Is Not Required

There is a common misperception that unless design standardization is achieved globally throughout the entire enterprise, SOA will not succeed. Although design standardization is a critical success factor for SOA projects that is *ideally* achieved across an enterprise, it only needs to be realized to a meaningful extent for service-orientation to result in strategic benefit.

For example, service-orientation emphasizes the need for standardizing service data models to avoid unnecessary data transformation and other problematic issues that can compromise interoperability. The extent to which data model standardization is achieved determines the extent to which these problems will be avoided.

The goal is not always to eliminate problems entirely because that can be an unrealistic objective, especially in larger enterprises. Therefore, the goal is sometimes to just minimize problems by taking special considerations into account during service design.

In support of this approach, design patterns exist for organizing the division of an enterprise into more manageable domains. Data standardization is generally more easily attained within each domain, and transformation is then only required when exchanging data across these domains. Even though this does not achieve a global data model, it can still help establish a very meaningful level of interoperability.

Reuse Is Not an Absolute Requirement

Increasing reusability of solution logic is a fundamental goal of service-orientation, and reuse is clearly one of the most associated benefits of SOA. As a result, organizations that have had limited success with past reuse initiatives, or with concerns that significant amounts of reuse cannot be achieved within their enterprise, are often hesitant about SOA in general.

While reuse, especially over time, can be one of the most rewarding parts of investing in SOA, it is not the sole primary benefit. Perhaps even more fundamental to service-orientation than promoting reuse is fostering interoperability. Enabling an enterprise to connect previously disparate systems or to make interconnectivity an intrinsic quality of new solution logic is extremely powerful.

You could ignore the principle of Service Reusability in service designs and still achieve significant returns on investment based solely on raising the level of enterprise-wide interoperability.

NOTE

One could argue that reuse and interoperability are very closely related in that if two services are interoperable, there is always the opportunity for reuse. However, traditional perspectives of reusable solution logic focus on the nature of the logic itself. A service that is designed to be specifically agnostic to business processes and cross-cutting to address multiple concerns will have a particular functional context associated with it. Therefore, reuse can be seen as a separate design characteristic that relies and builds upon interoperability. See Chapter 9 for more details.

SUMMARY OF KEY POINTS

- Service-orientation has deep roots in several past computing platforms and design approaches, and is therefore not considered a revolutionary design paradigm.
 - Global standardization within an enterprise is not a requirement for creating service-oriented enterprises because individual service inventories can be established (and separately standardized) within different enterprise domains.
 - Although fundamental to much of service-orientation, if reusability were to be omitted as a design characteristic, significant interoperability-related benefit would still be attainable.
-

4.5 Effects of Service-Orientation on the Enterprise

There are good reasons to have high expectations from the service-orientation paradigm. But, at the same time, there is much to learn and understand before it can be successfully applied. The following sections explore some of the more common examples.

Service-Orientation and the Concept of “Application”

Having just stated that reuse is not an absolute requirement, it is important to acknowledge the fact that service-orientation does place an unprecedented emphasis on reuse. By establishing a service inventory with a high percentage of reusable and agnostic services, we are now positioning those services as the primary (or only) means by which the solution logic they represent can and should be accessed.

As a result, we make a very deliberate move away from the silos in which applications previously existed. Because we want to share reusable logic whenever possible, we automate existing, new, and augmented business processes through service composition. This results in a shift where more and more business requirements are fulfilled not by building or extending applications, but by simply composing existing services into new composition configurations.

When compositions become more common, the traditional concept of an application, a system, or a solution actually begins to fade, along with the silos that contain them. Applications no longer consist of self-contained bodies of programming logic responsible for automating a specific set of tasks (Figure 4.15). What was an application is now just another service composition. And it’s a composition made up of services that very likely participate in other compositions (Figure 4.16).

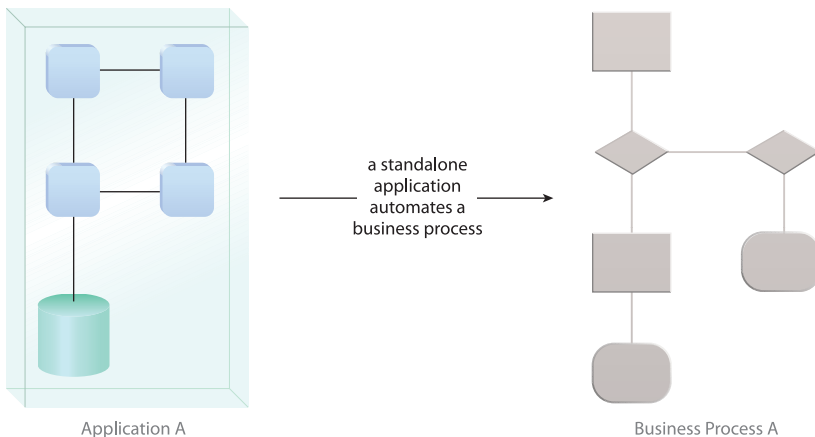


Figure 4.15

The traditional application, delivered to automate specific business process logic.

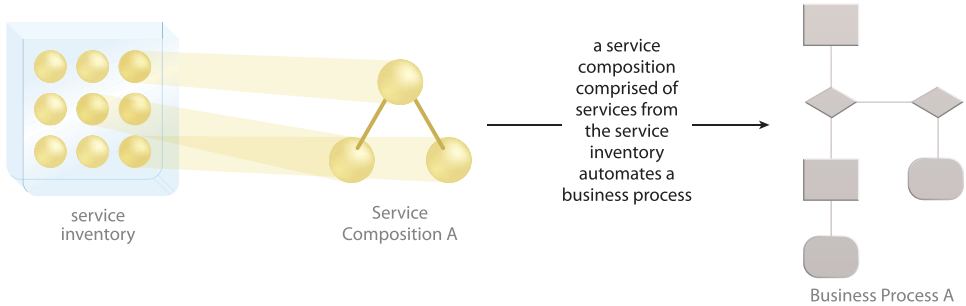


Figure 4.16

The service composition, intended to fulfill the role of the traditional application by leveraging agnostic and non-agnostic services from a service inventory. This essentially establishes a “composite application.”

An application in this environment loses its individuality. One could argue that a service-oriented application actually does not exist because it is, in fact, just one of many service compositions. However, upon closer reflection, we can see that some of the services are actually not business process-agnostic. The task service, for example, intentionally represents logic that is dedicated to the automation of just one business task and therefore is not necessarily reusable.

What this indicates is that non-agnostic services can still be associated with the notion of an application. However, within service-oriented computing, the meaning of this term can change to reflect the fact that a potentially large portion of the application logic is no longer exclusive to the application.

Service-Orientation and the Concept of “Integration”

When we revisit the idea of a service inventory consisting of services that have, as per our service-orientation principles, been shaped into standardized and (for the most part) reusable units of solution logic, we can see that this can challenge the traditional perception of “integration.”

In the past, integrating something implied connecting two or more applications or programs that may or may not have been compatible (Figure 4.17). Perhaps they were based on different technology platforms or maybe they were never designed to connect with anything outside of their own internal boundary. The increasing need to hook up disparate pieces of software to establish a reliable level of data exchange is what turned integration into an important, high profile part of the IT industry.

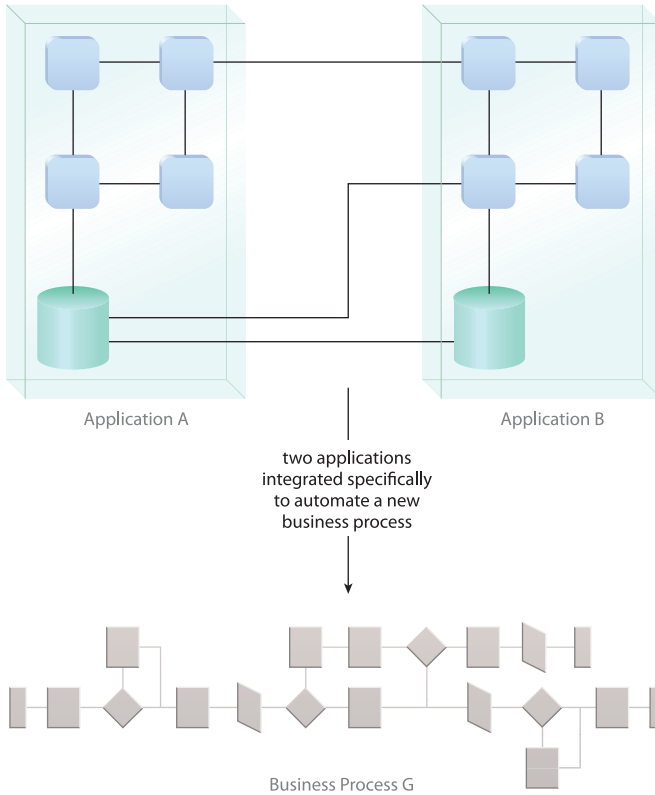


Figure 4.17
The traditional integration architecture, comprised of two or more applications connected in different ways to fulfill a new set of automation requirements (as dictated by the new Business Process G).

Services designed to be “intrinsically interoperable” are built with the full awareness that they will need to interact with a potentially large range of service consumers, most of which will be unknown at the time of their initial delivery. If a significant part of our enterprise solution logic is represented by an inventory of intrinsically interoperable services, it empowers us with the freedom to mix and match these services into infinite composition configurations to fulfill whatever automation requirements come our way.

As a result, the concept of integration begins to fade. Exchanging data between different units of solution logic becomes a natural and secondary design characteristic (Figure 4.18). Again, though, this is something that can only transpire when a substantial percentage of an organization’s solution logic is represented by a quality service inventory.

While working toward achieving this environment, there will likely be many requirements for traditional integration between existing legacy systems and also between legacy systems and these services.

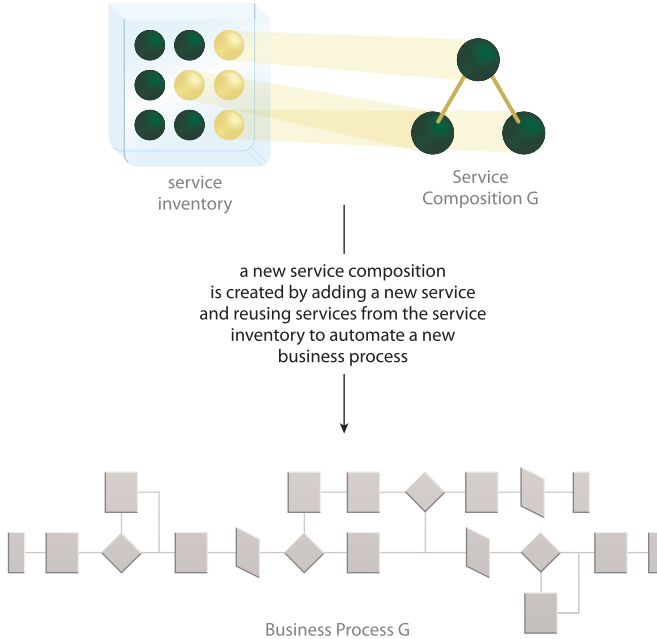


Figure 4.18
A new combination of services is composed together to fulfill the role of traditional integrated applications.

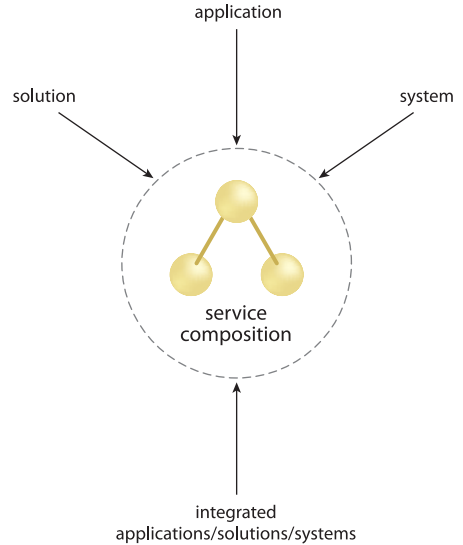
The Service Composition

Applications, integrated applications, solutions, systems, all of these terms and what they have traditionally represented can be directly associated with the service composition (Figure 4.19). However, given the fact that many SOA implementations consist of a mixture of legacy environments and services, these terms are sure to survive for quite some time.

In fact, as SOA transition initiatives continue to progress within an enterprise, it can be helpful to make a clear distinction between a traditional application (one which may reside alongside an SOA implementation or which may be actually encapsulated by a service) and the service compositions that eventually become more commonplace.

Figure 4.19

A service-oriented solution, application, or system is the equivalent of a service composition. If we were to build an enterprise-wide SOA from the ground up, it would likely be comprised of numerous service compositions capable of fulfilling the traditional roles associated with these terms.



Application, Integration, and Enterprise Architectures

Because applications have existed for as long as IT, when technology architecture as a profession and perspective within the enterprise came about, it made perfect sense to have separate architectural views dedicated to individual applications, integrated applications, and the enterprise as a whole.

When standardizing on service-orientation, the manner in which we document technology architecture is also in for a change. The enterprise-level perspective becomes predominant as it represents a master view of the service inventory. It can still encompass the traditional parts of a formal architecture, including conceptual views, physical views, and supporting technologies and governance platforms—but all these views are likely to now become associated with the service inventory.

A new type of technical specification that gains prominence in service-oriented enterprise initiatives is the *service composition architecture*. Even though we talk about the simplicity of combining services into new composition configurations on demand, it is by no means an easy process. It is a design exercise that requires the detailed documentation of the planned composition architecture.

For example, each service needs to be assessed as to its competency to fulfill its role as a composition member, and foreseeable service activity scenarios need to be mapped out.

Message designs, messaging routes, exception handling, cross-service transactions, policies, and many more considerations go into making a composition capable of automating its designated business process.

BEST PRACTICE

Although the structure and content of traditional application architecture specifications are augmented when documenting composition architectures, there can still be a natural tendency to refer to these documents as architecture specifications for applications.

While an organization is undergoing a transition toward SOA, it can be helpful to make a clear distinction between an application consisting of a service composition and traditional, standalone or legacy applications.

One approach is to consistently qualify the term “application.” For example, it can be prefixed with “service-oriented,” “composite,” “standalone,” or “legacy.” Another option is to simply limit the use of the term “application” to refer to non-service-composed solutions only.

Furthermore, a composed service encapsulating a legacy application can be documented in separate specifications: a composition architecture specification that identifies the service and points to an application architecture specification that defines the corresponding application.

SUMMARY OF KEY POINTS

- The traditional concept of an application can change as more agnostic services become established parts of the enterprise.
- The traditional concept of integration can change as the proliferation of standardized, intrinsic interoperable services increases.
- Architectural views of the enterprise shift in response to the adoption of service-orientation. Principally, the enterprise perspective becomes increasingly prominent.

4.6 Origins and Influences of Service-Orientation

It is often said that the best way to understand something is to gain knowledge of its history. Service-orientation, by no means, is a design paradigm that just came out of nowhere. It is very much a representation of the evolution of IT and therefore has many

roots in past paradigms and technologies (Figure 4.20). At the same time, it is still in a state of evolution itself and therefore remains subject to influences from on-going trends and movements.

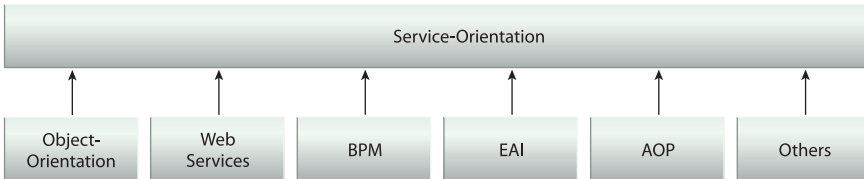


Figure 4.20

The primary influences of service-orientation also highlight its many origins.

The sections that follow describe some of the more prominent origins and thereby help clarify how service-orientation can relate to and even help further some of the goals from past paradigms.

Object-Orientation

In the 1990s the IT community embraced a design philosophy that would lead the way in defining how distributed solutions were to be built. This paradigm was object-orientation, and it came with its own set of principles, the application of which helped ensure consistency across numerous environments. These principles defined a specific type of relationship between units of solution logic classified as objects, which resulted in a predictable set of dynamics that ran through entire solutions.

Service-orientation is frequently compared to object-orientation, and rightly so. The principles and patterns behind object-oriented analysis and design represent one of the most significant sources of inspiration for this paradigm.

In fact, a subset of service-orientation principles (Service Reusability, Service Abstraction, and Service Composability, for example) can be traced back to object-oriented counterparts. What distinguishes service-orientation, though, are the parts of the object-oriented school of thought that were left out and the other principles that were added. See Chapter 14 for a comparative analysis of principles and concepts associated with these two design approaches.

Web Services

Even though service-orientation as a paradigm and SOA as a technology architecture are each implementation-neutral, their association with Web services has become commonplace—so much so that the primary SOA vendors have shaped their respective platforms around the utilization of Web services technology.

Although service-orientation remains a fully abstract paradigm, it is one that has historically been influenced by the SOA platforms and roadmaps produced by these vendors. As a result, the Web services framework has influenced and promoted several service-orientation principles, including Service Abstraction, Service Loose Coupling, and Service Composability.

Business Process Management (BPM)

BPM places a significant emphasis on business processes within the enterprise both in terms of streamlining process logic to improve efficiency and also to establish processes that are adaptable and extensible so that they can be augmented in response to business change.

The business process layer represents a core part of any service-oriented architecture. From a composition perspective, it usually assumes the role of the parent service composition controller. The advent of orchestration technology reaffirmed this role from an implementation perspective.

A primary goal of service-orientation is to establish a highly agile automation environment fully capable of adapting to change. This goal can be realized by abstracting business process logic into its own layer, thereby alleviating other services from having to repeatedly embed process logic.

While service-orientation itself is not as concerned with business process reengineering, it fully supports process optimization as a primary source of change for which services can be recomposed.

Enterprise Application Integration (EAI)

Integration became a primary focal point in the late 90's, and many organizations were ill prepared for it. Numerous systems were built with little thought given to how data could be shared outside of the system boundary. As a result, point-to-point integration

channels were often created when data sharing requirements emerged. This led to well known problems associated with a lack of stability, extensibility, and inadequate interoperability frameworks.

EAI platforms introduced middleware that allowed for the abstraction of proprietary applications through the use of adapters, brokers, and orchestration engines. The resulting integration architectures were, in fact, more robust and extensible. However, they also became notorious for being overwhelmingly complex and expensive, as well as requiring long-term commitments to the middleware vendor's platform and roadmap.

The advent of the open Web services framework and its ability to fully abstract proprietary technology changed the face of integration middleware. Vendor ties could be broken by investing in mobile services as opposed to proprietary platforms, and organizations gained more control over the evolution of their integration architectures.

Several innovations that became popularized during the EAI era were recognized as being useful to the overall goals associated with building SOA using Web services. One example is the broker component, which allows for services using different schemas representing the same type of data to still communicate through runtime transformation. The other is the orchestration engine, which can actually be positioned to represent an entire service layer within larger SOA implementations. These parts of the EAI platform support several service-orientation principles, including Service Abstraction, Service Statelessness, Service Loose Coupling, and Service Composability.

Aspect-Oriented Programming (AOP)

A primary goal of AOP is to approach the separation of concerns with the intent of identifying specific concerns that are common to multiple applications or automation scenarios. These concerns are then classified as “cross-cutting,” and the corresponding solution logic developed for cross-cutting concerns becomes naturally reusable.

Aspect-orientation emerged from object-orientation by building on the original goals of establishing reusable objects. Although not a primary influential factor of service-orientation, AOP does demonstrate a common goal in emphasizing the importance of investing in units of solution logic that are agnostic to business processes and applications and therefore highly reusable. It further promotes role-based development, allowing developers with different areas of expertise to collaborate.

NOTE

The actual events and timeline associated with the emergence of SOA are documented in Chapter 4 of the book *Service-Oriented Architecture: Concepts, Technology, and Design*.

SUMMARY OF KEY POINTS

- Service-orientation represents a design paradigm that has its roots in several origins. It emphasizes successful and proven approaches and supplements them with new principles that leverage recent conceptual and technology innovation.
- Service-orientation, as a design paradigm, is comparable with object-orientation. In fact, several key object-oriented principles have persisted in service-orientation.
- The Web services technology platform is primarily responsible for the popularity of SOA and is therefore also a significant influence in service-orientation. Conversely, the rise of service-oriented computing has repositioned and formalized the Web services technology set from its original incarnation.

4.7 CASE STUDY BACKGROUND

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.

Below are brief descriptions of the service candidates shown in Figure 4.21 in relation to how they inter-relate based on their entity-centric functional contexts:

- 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 their 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 levels 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.

Note that all services shown are entity services, with the exception of Patent Sweep and Notifications, which are based on the utility service model. A task service is added in Part II.



Figure 4.21

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 inventory management of manufactured and purchased products. All of the displayed services are based on the entity service model, except for the bottom two, which are utility services.

This page intentionally left blank

Index

A

- absolute isolation, 309, 317
- abstract classes (OOAD), 461
 - designing service-oriented classes, 474
- Abstract Syntax Notation 1 (ASN.1), 128
- abstraction (OOAD), 463. *See also* Service Abstraction (principle)
- access control levels, 232-234
- accessor methods (OOAD), 454
- active state (state management), 335
- aggregates of services. *See* service compositions
- aggregation (OOAD), 471-472
- agile development, 87, 521
 - organizational agility versus, 63
 - service-orientation and, 87
 - Service Reusability design risks, 287
- agility. *See* organizational agility
- agnostic capability candidates, 523
- agnostic service references, 63
- agnostic services, 62, 82, 91, 407
 - reusable services versus, 268-269
 - service contracts, 144
 - Service Reusability, 268-269
- agnostic solution logic, increasing, 82
- alignment of business and technology.
 - See* business and technology domain alignment in service-oriented computing
- analysis phase, measuring service reusability in, 265-266
- analysis scope, defining, 522
- AOP (aspect-oriented programming), as an influence of service-orientation, 99, 448
- API (application programming interface), 48, 128, 174, 177, 213, 313
 - functional abstraction, 221
 - service contracts and, 129
- application architectures, 95-96
- application programming interface.
 - See* API
- application services. *See* utility services
- application-specific solution logic, reducing, 82-83
- applications
 - composite, 91-92
 - service compositions versus, 91-92
 - service-orientation and, 91-92
 - technology architectures, 95-96
- architects. *See* enterprise architects (role)

architecture. *See also* SOA
 (service-oriented architecture)
 application, 95-96
 client-server, 128, 165
 state management, 328
 defining, 520
 distributed, 128, 166
 state management, 329, 331
 enterprise, 80, 95-96
 integration, 81, 92-96, 182-184
 mainframe, 166
 point-to-point, 80, 405-406
 service composition, 96
 Service Statelessness design risks,
 349-350
 of Web services, 48-49, 166
 ASN.1 (Abstract Syntax Notation 1),
 128
 aspect-oriented programming. *See* AOP
 assertions. *See* policy assertions
 association (OOAD)
 comparison of object-orientation
 and service-orientation, 469-470
 designing service-oriented classes,
 474
 attachments (SOAP), 334
 attributes (objects), explained, 454
 attributes (OOAD), 473
 auditors. *See* enterprise design
 standards custodians (role)
 auto-generation (of service contracts),
 175, 178
 autonomy. *See also* Service Autonomy
 (principle)
 composition autonomy, 430
 data models and, 308-310
 databases and, 308-310
 governance and, 298-299
 service compositions and, 298, 314

B

base classes (OOAD), 461
 benefits of service-oriented computing.
 See service-oriented computing,
 goals and benefits
 best practices
 architecture dependency, 350
 building Web services, 151
 controlled access, 234
 discoverability meta
 information, 382
 Domain Inventory design
 pattern, 275
 encapsulated legacy
 environments, 318
 example of, 34
 explained, 34-35
 measuring consumer coupling, 192
 service composition performance
 limitations, 437
 service contract design risks, 150
 for service-orientation, 87
 bidirectional coupling, 165
 black box concept, 213, 227
 books, related, 4-5
 Web site, 16
 bottom-up processes, 518-519
 BPM (business process management),
 as an influence of service-
 orientation, 98, 448
 bridging products, 142
 business agility. *See* organizational
 agility
 business analysts, 522
 discoverability meta information
 and, 377
 role of, 53
 business and technology domain
 alignment in service-oriented
 computing, 60-61

business data (state management), 338
 business entity services. *See* entity services
 business logic. *See* core service logic in Web sites
 business models. *See* enterprise business models
 business process definition, explained, 397
 business process instance, explained, 397
 business process management. *See* BPM
 business process services. *See* orchestrated task services; task services
 business requirements fulfillment, as goal of object-orientation, 450-451
 business service candidates, 377
 business services. *See* entity services; task services

C

candidates. *See* service candidates
 capabilities
 granularity and, 116
 operations and methods versus, 115
 service compositions, 399-400
 services and, 69-70
 capability candidates. *See* service capability candidates
 capability granularity, 486
 explained, 116
 Service Composability and, 428
 service contracts, 143
 Service Loose Coupling principle and, 195-196
 Service Reusability and, 277

Capability Name (service profile field), 481
 capability profiles, structure of, 481-482
 case study
 background, 20-22, 66, 100-101, 119-121
 business process description, 119-121
 conclusion of, 514-515
 coupling in, 202-209
 preliminary planning, 101
 service abstraction levels, 244-252
 Service Autonomy in, 319-323
 Service Composability in, 439-441
 Service Discoverability in, 382-386
 Service Reusability in, 288-292
 Service Statelessness in, 351-359
 services in, 154
 Standardized Service Contract principle example, 154-161
 style, 20
 centralization
 Contract Centralization design pattern, 185, 195, 473, 530
 example of, 216-217
 Logic Centralization and, 272-273
 measuring consumer coupling, 191-192
 standardized coupling and, 185
 technology coupling, 189-190
 Logic Centralization design pattern, 185, 465, 468, 531
 Contract Centralization and, 272-273
 difficulty in achieving, 274-275
 as enterprise design standard, 272
 explained, 271
 standardized coupling and, 185
 Web services and, 274

- of policy assertions, 138-139
 - Schema Centralization design pattern, 135-137, 531
- characteristics. *See* design characteristics
- chorded circle symbol, explained, 13, 15-16
- classes (OOAD)
 - compared to service contracts, 453
 - service-oriented classes, 472-474
- client-server architectures, 165, 128
 - state management, 328
- coarse-grained design. *See* granularity
- code examples
 - capability expressed in IDL, 129
 - capability expressed in WSDL, 129
 - constraint granularity, 117
 - fine-grained XML schema simple type, 143
 - skeleton (coarse- and fine-grained)
 - operation definitions, 143
 - skeleton WSDL definition for coarse-grained service, 142
 - SOAP and WS-Addressing headers for state management, 337
 - standardized and non-standardized WSDL message definitions, 133
 - UDDI discoveryURL construct, 372
 - WS-BPEL composition logic, 431
 - WS-Coordination headers for state management, 338
 - WS-MetadataExchange and WS-Addressing, 373
- cohesion
 - comparison of object-orientation and service-orientation, 467
 - service granularity and, 467
- collective composability, explained, 400-401
- color, in symbols, 13
- commercial product design, 62, 276
 - abstraction and, 214
 - coupling and, 166
 - gold-plating versus, 267
 - meta abstraction types in, 227
 - measuring service reusability, 262, 264-265
 - risks associated with, 286-287
- communications quality, 365
- communications specialists. *See* technical communications specialists (role)
- complete reusability, 266, 487
- complex compositions. *See* complex service compositions
- complex service activities, 402
- complex service compositions, 406-407, 487
 - characteristics of, 410-411
 - preparation for, 411
 - service inventory evolution, 407, 409-410
- complexity, in traditional solution delivery, 80
- components, coupling and, 176-177
- composability. *See* Service Composability (principle)
- composition (OOAD), 470-471. *See also* service compositions; Service Composability (principle)
- composition autonomy, 430
 - Service Composability and, 430
- composition candidates. *See* service composition candidates
- composition controller capabilities, 394, 400
- composition controllers, 435, 487
 - explained, 398-401
 - service consumers as, 404

- composition initiators, 487
 - explained, 403-405
 - service consumers as, 404
- composition instances, 397
- composition member capabilities, 393, 400
- Composition Member Capabilities (service profile field), 481
- composition members, 487
 - design of. *See* Service Composability (principle)
 - explained, 398-401
 - Web service region of influence for, 395
- Composition Role (service profile field), 481
- composition sub-controllers, 487
- concise contract abstraction, 232, 487
- conflict symbol, 13
- constraint granularity, 486
 - explained, 117-118
 - Service Abstraction and, 239
 - Service Composability and, 428
 - service contracts, 143
 - Service Loose Coupling principle and, 195-196
 - Service Reusability and, 278
- consumer coupling
 - measuring, 191-192
 - Service Abstraction and, 192
 - Service Composability and, 191
 - service consumers, 48-49
 - as composition initiators and controllers, 404*
 - coupling and, 167*
 - coupling types, 181-192*
 - policy dependencies, 138*
- consumer-specific functional coupling, 180
- consumer-to-contract coupling, 185-191, 473, 486
 - risks with, 214
 - Web services and, 186
- consumer-to-implementation coupling, 182, 184, 486
 - integration architectures and, 182-184
- containers, objects as, 458
- content abstraction, 246
- context data (state management), 337-338
- context rules (state management), 337
- Contract Centralization design pattern, 185, 195, 473, 530
 - example of, 216-217
 - Logic Centralization and, 272-273
 - measuring consumer coupling, 191-192
 - standardized coupling and, 185
 - technology coupling, 189-190
- contract content abstraction levels, 231-232
- Contract Denormalization design pattern, 242, 305, 312, 530
 - service contract autonomy and, 304-305
- contract first design, 53, 131, 173, 194
- contract-to-functional coupling, 180, 486
 - indirect consumer coupling and, 188
- contract-to-implementation coupling, 177-179, 486
 - examples of, 177
 - indirect consumer coupling and, 189
 - service composability, 200

- contract-to-logic coupling, 174-175, 486
 - policies and, 179
 - Service Composability and, 199
- contract-to-technology coupling, 176-177, 486
 - direct consumer coupling and, 188
 - Service Composability and, 199
- contracts. *See* service contracts
- controlled access (access control level), 233-234, 487
- controller capabilities, 400
- controllers. *See* composition controllers
- core service logic in Web services, 48
- coupling. *See also* Service Loose Coupling (principle)
 - architectural, 168
 - auto-generation and, 175
 - in case study, 202-209
 - in client-service architectures, 165
 - commercial product design and, 166
 - compared to dependency, 165
 - data models and, 175
 - database tables and, 175
 - design principles, relationship with, 197-200
 - design risks, 200
 - logic-to-contract coupling*, 200-201
 - performance problems*, 201-202
 - design-time autonomy and, 181, 315-316
 - in distributed architectures, 166
 - explained, 164-165
 - integration architectures and, 182-184
 - mainframe and, 166
 - multi-consumer coupling requirements (Service Abstraction principle), 242
 - negative types, 193, 195
 - in object-orientation, 166
 - origins of, 165-166
 - performance, 202
 - policies and, 179
 - positive types, 193, 195
 - proprietary components and, 176-177
 - risks with, 214
 - Service Composability and, 191
 - service consumer coupling types, 181-182
 - consumer-to-contract coupling*, 185-191
 - consumer-to-implementation coupling*, 182, 184
 - Contract Centralization design pattern*, 185
 - measuring consumer coupling*, 191-192
 - service contract coupling types, 169-173
 - contract-to-functional coupling*, 180
 - contract-to-implementation coupling*, 177-179
 - contract-to-logic coupling*, 174-175
 - contract-to-technology coupling*, 176-177
 - logic-to-contract coupling*, 173-174
 - service granularity and, 195-196
 - service models and, 196-197
 - service-orientation and, 193-195
 - symbols for, 165
 - Web services and, 166
- coupling quality, 146
- cross-cutting functions, 313, 347
- CRUD, 44, 464

cultural issues, Service Reusability
design risks, 281-283
Custodian (service profile field), 482
Cutit Saws case study. *See* case study

D

data granularity, 486
explained, 116
Service Composability and, 428
service contracts, 143
Service Loose Coupling principle
and, 195-196
Service Reusability and, 278

data models

autonomy and, 308-310
contract-to-implementation
coupling and, 177-178
coupling and, 175
data granularity and, 116
example of coupling, 206
global, 136
logical, 52
service contracts and, 134-137
standardization, 50, 89, 134-137

data representation standardization, 134-137

case study, 155
data transformation, avoiding,
140-142
sample design standards, 155

data transformation

avoidance, 135-136, 140-142
design standards and, 135-136
performance issues, 140
problems, 140
standardization and, 140-142
Standardized Service Contract
principle and, 135-136, 140-142

databases

autonomy and, 308-310
contract-to-implementation
coupling and, 177-178
coupling and, 175
for state management, 329, 331,
339-343

dedicated controllers, 487

deferral. *See* state deferral

delegation (OOAD), 468-469. *See also* state delegation

delivery processes. *See* processes

delivery strategies. *See* processes

denormalization. *See also* normalization service contracts and, 301-305

dependency, coupling compared to, 165

design characteristics

example of, 27
explained, 27-28
implementation of, 111-114
importance of, 69
list of, 81
loose coupling, 166
regulation of, 111-114

design framework, 35-36

design granularity. *See* granularity

design paradigm

example of, 29
explained, 29-30
relationships with design
framework, 36
service-orientation as, 70-71

design pattern language

example of, 32
explained, 31-32

design patterns

- Contract Centralization design pattern, 185, 195, 242, 473, 530
 - example of, 216-217*
 - Logic Centralization and, 272-273*
 - measuring consumer coupling, 191-192*
 - standardized coupling and, 185*
 - technology coupling, 189-190*
- Contract Denormalization, 242, 305, 312
 - service contract autonomy and, 304-305*
- Domain Inventory, 136, 275
 - example of, 31*
 - explained, 30-31*
 - how they are referenced, 111*
- Logic Centralization, 185, 465, 468
 - Contract Centralization and, 272-273*
 - difficulty in achieving, 274-275*
 - as enterprise design standard, 272*
 - explained, 271*
 - Web services and, 274*
- referenced in design principles, 530
- relationships with design framework, 36
- Schema Centralization, 135-137
- Service Normalization, 272, 305, 465
 - service contract autonomy and, 302-304*

design phase (service composition), 413
 assessment, 413, 415

design principles

- application levels, vocabularies for, 487-488
- best practices versus, 34

- business and technology alignment in, 502-503
- compared to object-oriented design principles, 457-472
- design pattern references, 111, 530
- design standards and, 33, 107-108
- documentation for, 109-110
- example of, 28*
- explained in abstract, 28-29*
- extent of implementation, 108*
- federation in, 501*
- in formal service design processes, 106-107*
- granularity, types of, 115-118*
- guidelines for working with, 104-110, 115-121*
- implementation mediums and, 114-115*
- implementation of design characteristics, 111-114*
- interoperability and, 74-75*
- intrinsic interoperability in, 498, 500*
- list of, 71-73*
- mapping to strategic goals, 498-509*
- organizational agility in, 505, 507*
- principle profiles, explained, 109-110*
- reduced IT burden in, 507, 509*
- regulation of design characteristics, 111-114*
- ROI in, 504*
- Service Abstraction, relationship with, 239-241. See also Service Abstraction (principle)*
- Service Autonomy, relationship with, 314-317. See also Service Autonomy (principle)*

- Service Composability,
 - relationship with, 432-436. *See also* Service Composability (principle)
 - service contracts. *See* service contracts
 - Service Coupling (principle),
 - relationship with, 197-200
 - Service Discoverability,
 - relationship with, 378-380. *See also* Service Discoverability (principle)
 - Service Reusability (principle),
 - relationship with, 278, 280-281
 - Service Statelessness, relationship with, 347-349. *See also* Service Statelessness (principle)
 - in service-oriented analysis, 105-106
 - service-oriented computing elements, relationship with, 41
 - SOA goals and benefits,
 - relationship with, 498-499
 - standard structure, 109-110
 - standardization of service contracts, relationship with, 144-148
 - vendor diversification in, 501-502
 - vocabularies for, 486-487
- design standards**
- data representation design
 - standard samples, 155
 - design principles and, 107-108
 - example of, 33
 - explained, 32-33
 - functional expression design
 - standard samples, 155
 - granularity and, 144
 - importance of, 86
 - industry standards versus, 34
 - level required, 89
 - naming conventions, 147
 - in service-orientation, 86
 - Standardized Service Contract
 - principle and, 132
- design taxonomy, 35**
- design-time autonomy, 486**
- coupling and, 315-316
 - explained, 298-299
 - logic-to-contract coupling and, 181
 - service contracts and, 301-305
- design-time discovery, 371-373, 486**
- design-time isolation, 309**
- designated controllers, explained, 400**
- detailed contract abstraction level, 231, 487**
- development tool deficiencies, 151-152**
- direct consumer coupling**
- example of, 188
 - indirect consumer coupling versus, 186, 188-189
- discoverability, explained, 364. *See also* Service Discoverability (principle)**
- discovery. *See also* Service Discoverability (principle)**
- explained, 364-366
 - meta information and, 362
 - origins of, 367-368
 - processes, 363-367
 - of resources, 362-368
 - types of, 371-373
- distributed architectures, 128, 166**
- state management, 329, 331
- DLL (dynamic link library), 390**
- document-centric messages, 117**
- Domain Inventory design pattern, 136, 275, 531**
- don't repeat yourself. *See* DRY (OOAD)**
- DRY (OOAD), 465-466**
- dynamic link library. *See* DLL**

E

EAI, 213, 448
 as an influence of service-orientation, 98-99, 448

encapsulation
 of legacy logic, 318
 Service Abstraction versus, 235
 service encapsulation, 235-237

encapsulation (OOAD), 458

Endpoint References, 345

enterprise application integration.
See EAI

enterprise architects (role), 494-495

enterprise architectures, 80, 95-96

enterprise business models,
 defining, 520

enterprise design standards custodians
 (role), 495

entity schemas, 136

entity services
 coupling and, 196
 design processes, 526
 example of, 44
 explained, 44
 Service Abstraction principle, 239
 Service Autonomy and, 312-313
 service contracts, 144
 Service Statelessness and, 346

entity-centric business services. *See*
 entity services

entity-centric schemas, 137

errata, 16

event-driven, 48

examples. *See* case study; code
 examples; For Example sections

extends attribute, 460

extensibility, as goal of object-orientation, 450-451

F

façade classes (OOAD), designing
 service-oriented classes, 474

federated service architecture, 59

federation
 in service-oriented computing,
 58-59
 with services, 58
 Web services and, 59

fine-grained design. *See* granularity

first-generation Web services platform,
 47. *See also* Web services

flexibility, as goal of object-orientation,
 450, 452

For Example sections
 composition initiators, 404-405
 contract-to-implementation
 coupling, 179
 contract-to-logic coupling, 175
 contract-to-technology
 coupling, 177
 design standards, 108
 formal service design
 processes, 107
 logic-to-contract coupling, 174
 messaging, 344
 Service Abstraction principle,
 216-217
 service contract autonomy, 303
 service modeling process, 106
 Service Reusability, 284-285
 XML schema standardization, 137

fully deferred state management,
 measuring service statelessness,
 342-343

functional abstraction, 221-222, 225, 486
 example of, 246

functional context, 70, 312, 468
 service granularity and, 116

functional coupling. *See* contract-to-functional coupling

functional expression
standardization, 155

functional isolation, 308

functional meta data, 374, 486
example of, 383-386

functional scope, Service Autonomy
design risks, 317

functional service expression,
standardization of, 133-134
case study, 155

fundamental concepts, comparison of
object-orientation and service-orientation, 453-454, 456-457

G

generalization (OOAD), 461-462

global data models, 136

glossary Web site, 16, 533

goals

comparison of object-orientation
and service-orientation, 449-452
mapping to design principles,
498-509

goals of service-oriented computing.
See service-oriented computing,
goals and benefits

gold-plating, 267

governance

autonomy and, 298-299, 316
design-time autonomy and,
298-299

pure autonomy, 308

reuse and, 316

Service Composability design
risks, 438

Service Reusability design risks,
283-285

of service-orientation, 88

governance phase (service
composition), 413

assessment, 417, 419

granularity. *See also* capability

granularity; constraint granularity;
data granularity; service granularity

design standards and, 144

levels, 118

types of, 115-118

Guidelines for Policy Assertion Authors
(W3C), 493

H

hardware accelerators, 334

has-a relationships (OOAD),
469-471, 474

hidden compositions, 402, 434

hiding information. *See* Service
Abstraction (principle)

high statelessness, 342-343

history. *See* origins

I

IDL (Interface Definition
Language), 128

implementation coupling, example of,
206-207

implementation mediums, design
principles and, 114-115

implementation phase, measuring
service reusability in, 267

implementation principles, 111-114

implementation requirement, service
contracts, 131

increased intrinsic interoperability, 75

indirect consumer coupling

direct consumer coupling versus,
186, 188-189

example of, 188-189, 207

- industry standards, design standards
 - versus, 34. *See also* Web services
 - information architecture models, 52
 - information hiding. *See* Service Abstraction (principle)
 - infrastructure services. *See* utility services
 - inheritance (OOAD), 166
 - comparison of object-orientation and service-orientation, 459-460
 - designing service-oriented classes, 473
 - service granularity and, 473
 - Input/Output (service profile field), 481
 - integration
 - of architectures, 81
 - consumer-to-implementation coupling, 182-184
 - coupling and, 182-184
 - EAI (enterprise application integration), 98-99
 - service compositions and, 92-94
 - service-orientation and, 84, 92-94
 - in traditional solution delivery, 80-81
 - integration architectures, 95-96
 - Interface Definition Language. *See* IDL
 - interface element, 456
 - interfaces (OOAD)
 - compared to service contracts, 456-457
 - compared to WSDL portType and interface elements, 456
 - designing service-oriented classes, 473
 - measuring service statelessness, 342
 - interoperability
 - of services, 84
 - service-orientation and, 74-75, 84
 - in service-oriented computing, 56-57
 - interpretability. *See also* Service Discoverability (principle)
 - defined, 365
 - explained, 365
 - interpretation process, 364-367
 - explained, 365
 - intrinsic interoperability. *See* interoperability
 - inventory analysis, 520-521, 523
 - is-a relationships (OOAD), 459
 - is-a-kind-of relationships (OOAD), 461
 - isolation
 - levels of, 308-310
 - partially isolated services, 306-308
 - of services, 308-310
 - IT roles. *See* organizational roles
- J–K**
- JDBC, 166
 - Keywords (service profile field), 481
- L**
- LDAP directories, 367
 - legacy systems
 - effect on, 523
 - mainframe architectures, 166
 - Service Autonomy design risks, 318
 - service encapsulation, 236
 - lifecycle phases of service
 - composability, 412-413
 - logic abstraction. *See* programmatic logic abstraction

Logic Centralization design pattern, 185, 465, 468, 531
 Contract Centralization and, 272-273
 difficulty in achieving, 274-275
 as enterprise design standard, 272
 explained, 271
 standardized coupling and, 185
 Web services and, 274

Logic Description (service profile field), 481

logic-to-contract coupling, 173-174, 486
 design-time autonomy and, 181
 example of, 174
 limitations, 200-201
 Web services and, 201

logic-to-implementation coupling, 178

logical data models, 52

loose coupling. *See* Service Loose Coupling (principle)

low-to-no statelessness, 340

M

mainframe architectures, 166

measuring
 consumer coupling, 191-192
 Service Abstraction, 231
access control abstraction levels, 232-234
contract content abstraction levels, 231-232
quality of service meta information, 234

Service Autonomy, 300-301
mixed autonomy, 310
pure autonomy, 308-310
service contract autonomy, 301-305
service logic autonomy, 306-308
shared autonomy, 305-306

Service Composability, 412
checklists, 419-420, 426-427
design phase assessment, 413, 415
governance phase assessment, 417, 419
lifecycle phases, 412-413
runtime phase assessment, 415, 417

Service Discoverability
baseline measures checklist, 375-376
custom measures, 376

Service Reusability, 262-263
in analysis/design phase, 265-266
commercial design approach, 262, 264-265
gold-plating, 267
in implementation phase, 267

Service Statelessness, 339
fully deferred state management, 342-343
internally deferred state management, 342
non-deferred state management, 340
partially deferred memory, 340-341
partially deferred state management, 341-342

message correlation, 337

message processing logic for Web services, 48

messages. *See also* SOAP
 comparison of object-orientation and service-orientation, 454-456
 data granularity and, 116
 document-centric, 117
 RPC-style, 117
 as state deferral option, 343-344

meta abstraction types, 218-219
 in commercial software, 227
 in custom-developed software, 228-229
 functional abstraction, 221-222
 in open source software, 227-228
 programmatic logic abstraction, 222-223
 quality of service abstraction, 224
 technology information
 abstraction, 219-221
 Web service design and, 225-226
 in Web services, 229-230

meta information types. *See* Service Discoverability (principle)

methods (objects), explained, 454

mixed autonomy, 310, 313

mixed detailed contract abstraction level, 232, 487

moderate statelessness, 341-342

modularization of policy assertions, 138-139

monolithic executables, 390

multi-consumer coupling requirements (Service Abstraction principle), 242

multi-purpose logic, 268

multi-purpose programs, 255-256

multi-purpose services, 468

N

naming conventions. *See* vocabularies

negative types of coupling, 193, 195

nested policy assertions, 138

.NET, 177, 216-217

no access (access control level), 234, 487

non-agnostic capability candidates, 523

non-deferred state management, 340

non-technical service contracts, 152-153. *See also* SLA
 Service Abstraction and, 237-238

normalization
 Contract Denormalization design pattern, 305, 312, 530
service contract autonomy and, 304-305
 entity services, 313
 service contracts and, 301-305
 Service Normalization design pattern, 272, 305, 465, 531
service contract autonomy and, 302-304
 of services, 65, 83
 utility services, 313

notification service for updates to *Prentice Hall Service-Oriented Computing Series* from Thomas Erl books, 17, 533

O

object-orientation, 129
 abstract classes, 461
designing service-oriented classes, 474
 abstraction, 213, 463. *See also* Service Abstraction (principle)
 accessor methods, 454
 aggregation, 471-472
 association
comparison of object-orientation and service-orientation, 469-470
designing service-oriented classes, 474
 attributes, 473
 base classes, 461

- classes
 - compared to service contracts*, 453
 - service-oriented classes*, 472-474
- composition, 470-471. *See also* service compositions; Service Composability (principle)
- coupling, 166
- delegation, 468-469. *See also* state delegation
- as design paradigm, 30
- DRY, 465-466
- encapsulation, 458
- façade classes, designing service-oriented classes, 474
- generalization, 461-462
- has-a relationships, 469-471, 474
- as influence of Service Composability, 391
- as influence of service-orientation, 97
- inheritance, 166
 - comparison of object-orientation and service-orientation*, 459-460
 - designing service-oriented classes*, 473
 - service granularity and*, 473
- interfaces
 - compared to service contracts*, 456-457
 - compared to WSDL portType and interface elements*, 456
 - designing service-oriented classes*, 473
 - measuring service statelessness*, 342
- is-a relationships, 459
- is-a-kind-of relationships, 461
- OCP, 465
- polymorphism, 463-464
- reuse and, 257
- RPC, 448
- service-orientation compared, 97, 446-475
 - common goals*, 449-452
 - design principles*, 457-472
 - fundamental concepts*, 453-457
- specialization, 461-462
- SRP, 466-468
- sub-classes, 459, 461, 463
- super-classes, 459
- uses-a relationships, 469, 471, 474
- object-oriented design principles, compared to service-orientation design principles, 457-458, 460-471
- objects
 - compared to services, 453
 - as containers, 458
- OCP (OOAD), 465
- ODBC, 166
- ontologies, 52
- OOAD (object-oriented analysis and design). *See* object-orientation
- open access (access control level), 233, 487
- open source software, meta abstraction types in, 227-228
- open-closed principle. *See* OCP
- optimized contract abstraction level, 232, 487
- orchestrated task services
 - coupling and, 197
 - defined, 45
 - Service Abstraction principle, 239
 - Service Autonomy and, 313-314

Service Composability and, 430, 432

Service Statelessness and, 347

orchestration. *See* orchestrated task services; WS-BPEL

orchestration services. *See* orchestrated task services

organizational agility

- agile development versus, 63
- project delivery timelines and, 64
- responsiveness and, 63
- Service Abstraction principle support for, 506
- service compositions and, 64
- Service Loose Coupling principle support for, 506
- Service Reusability principle support for, 64, 506
- service-orientation and, 63
- in service-oriented computing, 63-64

organizational culture. *See* cultural issues

organizational roles, 488-490

- enterprise architects, 494-495
- enterprise design standards custodians, 495
- policy custodians, 493
- schema custodians, 492
- service analysts, 491
- service architects, 491
- service custodians, 492
- service registry custodians, 493-494
- technical communications specialists, 494

origins

- of autonomy, 295
- of composition, 390-392
- of coupling, 165-166

- of discovery, 367-368
- of information hiding, 213
- of reuse, 257-258
- of service-orientation, 96-99
 - AOP (aspect-oriented programming)*, 99
 - BPM (business process management)*, 98
 - EAI (enterprise application integration)*, 98-99
 - object-orientation*, 97
 - Web services*, 98
- of service contracts, 127-129
- of state management, 328-331

overestimating service usage requirements, 318

P

paradigm. *See* design paradigm

parameters in policy assertions, 138

parent process coupling, 180

partially deferred memory, 340-341

partially deferred state management, 341-342

partially isolated services, 306-308

passive state (state management), 335

pattern languages. *See* design pattern languages

patterns. *See* design patterns

performance

- data transformation, 140
- schema coupling and, 202
- Service Composability design risks, 437-438
- service loose coupling, 201-202
- state management and, 334

Plain Old XML. *See* POX

planned reuse, measures of, 265-266

- point-to-point data exchanges,
 - explained, 80, 405-406
- policies, 48, 137-139, 274, 493
 - centralization and, 138
 - contract-to-logic coupling, 179
 - editors, 152
 - processors, 138
 - Service Abstraction and, 238
 - service consumer dependencies and, 138
 - service profiles and, 483
 - structural standards, 139
- policy alternatives, 378
- policy assertions, 146, 493
 - centralization, 138-139
 - modularization, 138-139
 - nested policy assertions, 138
 - parameters, 138
 - proprietary vocabularies for discoverability, 378
 - Service Discoverability and, 378
 - structural design, 139
 - structural standards and, 139
 - vocabularies for, 137-138
- policy custodians (role), 493
- policy parameters, 378
- policy vocabularies, 493
- polymorphism (OOAD), 463-464
- portType element, 456
- positive types of coupling, 193, 195
- post-implementation application of service discoverability, 381
- poster Web site, 16, 534
- POX (Plain Old XML), 50
- Prentice Hall Service-Oriented Computing Series from Thomas Erl*, 4, 111, 284, 495, 531
 - Web site, 16, 533
- primitive compositions, 406, 487
- primitive service activities, 402, 405
- principle profiles
 - explained, 109-110
 - Service Abstraction, 214-217
 - Service Autonomy, 296-297
 - Service Composability, 392, 395-396
 - Service Discoverability, 368, 370
 - Service Loose Coupling, 167, 169
 - service profiles versus, 110
 - Service Reusability, 259-261
 - Service Statelessness, 331-332, 334
 - Standardized Service Contract, 130-132
- principles. *See* design principles
- privacy concerns, Service Abstraction principle, 243
- process services. *See* orchestrated task services
- process-specific services, service contracts for, 144
- processes
 - bottom-up, 518-519
 - choosing, 521-522
 - discovery, 363-367
 - interpretation, 364-367
 - inventory analysis cycle, 520-521
 - service delivery, 518, 521-528
 - service modeling, 105-106, 523
 - service-oriented analysis, 105-106, 521
 - service-oriented design, 106-107
 - SOA delivery, 518, 521-528
 - top-down, 518-519
- productivity, as goal of object-orientation, 450, 452
- profiles. *See* principle profiles; service profiles

programmatic logic abstraction,
222-223, 226, 486

proprietary assertion vocabularies, 378

proprietary vocabularies, 137-138

proxies, 128

pure autonomy, 308-310, 317, 488

Purpose Description (service profile
field), 481

Q

QoS Requirements (service profile
field), 481

quality of service abstraction, 224,
226, 486

quality of service meta information,
374, 486

- abstraction levels and, 234
- example of, 386

R

reduced IT burden, as supported by
Service Composability principle, 509

reduced statefulness, 340-341

redundancy

- avoidance of, 64, 465-466
- reducing, 83
- in silo-based applications, 78
- in traditional solution delivery,
78-79

registries. *See* service registries

regulatory presence, 241

regulatory principles, 111-114

reliability, 317

- Service Reusability design
risks, 286

repository versus registry, 367

REST (Representational State
Transfer), 50

return on investment. *See* ROI

reusability, 69. *See also* Service
Reusability (principle)

- as goal of object-orientation,
450, 452
- level required, 90
- reuse versus, 256

reusable components (Standardized
Service Contract principle), 129

reuse, 62-63, 69, 82, 90. *See also* Service
Reusability (principle)

- explained in abstract, 254-256
- governance rigidity and, 438
- origins of, 257-258
- reusability versus, 256
- traditional approaches, 258
- traditional problems with, 257-258
- Web services and, 258

risks

- with consumer-to-contract
coupling, 214
- of gold-plating, 267
- Service Abstraction design, 242
 - human misjudgment*, 242-243
 - multi-consumer coupling
requirements*, 242
 - security and privacy
concerns*, 243
- Service Autonomy design
 - functional scope*, 317
 - overestimating service usage
requirements*, 318
 - wrapper services*, 318
- Service Composability design
 - governance rigidity*, 438
 - performance limitations*,
437-438
 - single points of failure*, 437

- Service Contract design, 149
 - development tool deficiencies, 151-152*
 - technology dependencies, 150*
 - versioning, 149-150*
 - Service Discoverability design
 - communication limitations, 381-382*
 - post-implementation application, 381*
 - Service Loose Coupling design, 200
 - logic-to-contract coupling, 200-201*
 - performance problems, 201-202*
 - Service Reusability design, 281
 - agile delivery, 287*
 - commercial design, 286-287*
 - governance structure, 283-285*
 - organizational culture, 281-283*
 - reliability, 286*
 - security, 286*
 - Service Statelessness design
 - architecture dependency, 349-350*
 - runtime performance, 350*
 - underestimating effort requirements, 350*
 - robustness, as goal of object-orientation, 450-451
 - ROI (return on investment)
 - Service Composability principle support for, 505
 - Service Discoverability principle support for, 505
 - Service Statelessness principle support for, 505
 - in service-oriented computing, 61-62
 - roles. *See* organizational roles
 - RPC, 150, 448, 455
 - RPC-style messages, 117
 - runtime autonomy, 486
 - explained, 298
 - normalization design patterns, 305
 - service contracts and, 301-305
 - runtime discovery, 371-373, 486
 - runtime performance (Service Statelessness design risks), 350
- S**
- scalability, 326, 333, 340, 348
 - Schema Centralization design pattern, 135-137, 531
 - schema custodians (role), 492
 - scope
 - of analysis, defining, 522
 - comparison of object-orientation and service-orientation, 447
 - second-generation Web services platform, 47. *See also* Web services
 - security
 - Service Abstraction principle, 243
 - Service Reusability design risks, 286
 - separation of concerns, 70
 - in relation to service compositions, 390
 - Service Abstraction (principle), 72, 212-251, 402
 - application level terminology, 487
 - associated terminology, 486
 - in case study, 244-252
 - commercial product design and, 214

- compared to abstraction (OOAD), 463
- considerations when designing service-oriented classes, 473
- constraint granularity and, 239
- consumer coupling and, 192
- contribution to realizing organizational agility, 506
- design principles, relationship with, 239-241
- design risks, 242
 - human misjudgment*, 242-243
 - multi-consumer coupling requirements*, 242
 - security and privacy concerns*, 243
- effect on other design principles, 239-241
- encapsulation versus, 235-237
- explained, 212
- goals, 215
- impact on composition design process, 418
- implementation requirements, 216
- interoperability and, 74
- measuring, 231
 - access control abstraction levels*, 232-234
 - contract content abstraction levels*, 231-232
 - quality of service meta information*, 234
- meta abstraction types, 218-219
 - in commercial software*, 227
 - in custom-developed software*, 228-229
 - functional abstraction*, 221-222
 - in open source software*, 227-228
 - programmatic logic abstraction*, 222-223
 - quality of service abstraction*, 224
 - technology information abstraction*, 219-221
 - Web service design and*, 225-226
 - in Web services*, 229-230
- non-technical contract documents and, 237-238
- origins of, 213
- policies and, 238
- policy assertions, 238
- principle profile, 214-217
- Service Autonomy and, 316
- Service Composability and, 241, 433-435
- Service Discoverability and, 241, 379
- service granularity and, 238-239
- Service Loose Coupling and, 114, 198, 241
- service models and, 239
- Service Reusability and, 241, 279
- Standardized Service Contract principle and, 146, 240
- Web services and, 50
- WS-Policy definitions, 238
- service activities, explained, 402-403, 487
- service adapters, 142, 174, 213
- service agents, 114
 - in message processing logic, 48
- service analysts (role), 491
- service architects (role), 491
- Service Autonomy (principle), 72, 276, 294-323
 - application level terminology, 488
 - associated terminology, 486
 - in case study, 319-323
 - composition autonomy and, 430

- considerations when designing service-oriented classes, 473
- coupling and, 178
- design principles, relationship with, 314-317
- design risks
 - functional scope*, 317
 - overestimating service usage requirements*, 318
 - wrapper services*, 318
- design-time autonomy, explained, 298-299
- effect on other design principles, 314-317
- explained, 294-295
- interoperability and, 74
- measuring, 300-301
 - mixed autonomy*, 310
 - pure autonomy*, 308-310
 - service contract autonomy*, 301-305
 - service logic autonomy*, 306-308
 - shared autonomy*, 305-306
- origins of, 295
- principle profile, 296-297
- runtime autonomy, explained, 298
- scalability, 261
- Service Abstraction and, 316
- Service Composability and, 317, 435-436
- service contracts, 301-305
- service granularity and, 311-312
- Service Loose Coupling and, 178, 199, 315-316
- service models and, 105, 311-314, 525
- Service Reusability and, 280, 316
- Service Statelessness and, 316, 348
- service-oriented analysis processes and, 105
- Standardized Service Contract and, 301-305, 315
- service candidates, 269, 276. *See also* service modeling
 - explained, 52
 - Service Discoverability and, 377
 - service inventory blueprint definition, 520
 - service modeling and, 52
 - service-oriented design and, 53
 - services versus, 52
- service capabilities
 - composition design support, assessment for, 422
 - composition governance support, assessment for, 426
 - composition runtime support, assessment for, 424
 - explained, 115
 - granularity and, 116
 - operations and methods versus, 115
- service capability candidates, 523, 525. *See also* service candidates
- service catalogs, service profiles and, 483
- Service Composability (principle), 73, 388-441. *See also* composition (OOAD)
 - associated terminology, 487
 - in case study, 439-441
 - composition autonomy and, 430
 - composition controllers, explained, 398-401
 - composition initiators, explained, 403-405
 - composition members, explained, 398-401
 - considerations when designing service-oriented classes, 473-474

- consumer coupling and, 191
- contract-to-implementation coupling and, 200
- contract-to-logic coupling and, 199
- contract-to-technology coupling and, 199
- contribution to realizing reduced IT burden, 509
- contribution to realizing ROI, 505
- design principles, relationship with, 432-436
- design risks
 - governance rigidity*, 438
 - performance limitations*, 437-438
 - single points of failure*, 437
- effect on other design principles, 432-436
- explained, 388
- interoperability and, 75
- measuring, 412
 - checklists*, 419-420, 426-427
 - design phase assessment*, 413, 415
 - governance phase assessment*, 417, 419
 - lifecycle phases*, 412-413
 - runtime phase assessment*, 415, 417
- orchestration and, 430, 432
- point-to-point data exchanges, explained, 405-406
- principle profile, 392, 395-396
- Service Abstraction and, 241, 433-435
- service activities, explained, 402-403
- Service Autonomy and, 317, 435-436
- service composition instances, explained, 397
- service compositions
 - capabilities*, 399-400
 - explained*, 397
- Service Discoverability and, 380, 436
- service granularity and, 427-428
- Service Loose Coupling and, 199-200, 433
- service models and, 428-430
- Service Reusability and, 280, 435
- Service Statelessness and, 436
- Standardized Service Contract and, 148, 432
- Web service region of influence, 395-396
- Web services and, 50, 401
- service composition candidates, 523
- service composition instances, explained, 397
- service composition references, 63
- service compositions, 82
 - agnostic services, 62
 - applications versus, 91-92
 - architecture of, 95-96
 - autonomy and, 298, 314
 - capabilities, 399-400
 - compared to applications and integrated applications, 94-95
 - complex service compositions, 407
 - characteristics of*, 410-411
 - preparation for*, 411
 - service inventory evolution*, 407, 409-410
 - composition autonomy, 430
 - consumer coupling and, 191
 - defined, 39
 - design assessment, 413

- evolutionary cycles, 412-413
 - design phase*, 413
 - governance phase*, 413
 - runtime phase*, 413
- explained, 39-40, 94-95, 388-390, 397
- governance assessment, 417
- governance considerations, 438
- hidden, 434
- implementation, 42
- integrated applications versus, 92-94
- naming, 96
- origins of, 390-392
- as related to service inventories, 407
- relationship with service-oriented computing elements, 40
- roles
 - composition controllers*, 398-399
 - composition initiators*, 403-404
 - composition members*, 398-399
 - designated controllers*, 400
 - examples of*, 404-405
- runtime assessment, 415, 417
- scope of, 405-406
- service contracts and, 148
- state management and, 340
- types of, 406
- service consumers, 48-49**
 - as composition initiators and controllers, 404
 - coupling and, 167
 - coupling types, 181-182
 - consumer-to-contract coupling*, 185-191
 - consumer-to-implementation coupling*, 182, 184
 - Contract Centralization design pattern*, 185
 - measuring consumer coupling*, 191-192
 - policy dependencies, 138
- service contract autonomy, 301-305, 488**
- service contracts, 126, 393. See also Standardized Service Contracts (principle)**
 - APIs and, 129
 - auto-generation, 54, 152
 - in client-service applications, 128
 - content abstraction levels, 231-232
 - data models and, 134-137
 - defined, 126
 - denormalization and, 301-305
 - dependencies on, 165
 - design-time autonomy and, 301-305
 - discoverability, 364-367
 - in distributed applications, 128
 - explained, 126-127
 - interpretability, 364-367
 - naming conventions, 133
 - non-technical contract documents, Service Abstraction and, 237-238
 - normalization and, 301-305
 - runtime autonomy and, 301-305
 - Service Autonomy and, 301-305
 - service compositions and, 148
 - technical versus non-technical, 127
 - validation coupling and, 190-191
 - versions, 150
 - Web services architecture, 48
- service coupling. See coupling**
- service custodians (role), 492**
- service description documents, explained, 126**
- service design**
 - capability granularity and, 116

- constraint granularity and, 117-118
 - data granularity and, 116
 - formal processes, design principles
 - in, 106-107
 - granularity levels, 118
 - granularity types, 118
 - normalization and, 65
 - separation of concerns and, 70
 - service granularity and, 116
 - Service Reusability principle
 - design principles, relationship with*, 278, 280-281
 - service granularity*, 277-278
 - service models*, 276-278
 - service-orientation principles and, 106-107
- Service Discoverability (principle)**, 73, 243, 272, 276, 362-386. *See also* discovery
- associated terminology, 486
 - in case study, 382-386
 - contribution to realizing ROI, 505
 - design principles, relationship with, 378-380
 - design risks
 - communication limitations*, 381-382
 - post-implementation application*, 381
 - discovery types, design-time and runtime discovery, 371-373
 - effect on other design principles, 378-380
 - explained, 362-364
 - implementation requirements, 370
 - interoperability and, 75
 - measuring
 - baseline measures checklist*, 375-376
 - custom measures*, 376
 - meta information types, 373
 - functional meta data*, 374
 - quality of service meta data*, 374
 - policy assertions and, 378
 - principle profile, 368, 370
 - Service Abstraction and, 241, 379
 - Service Composability and, 380, 436
 - service granularity and, 378
 - Service Loose Coupling and, 199
 - service modeling and, 106, 377-378, 525
 - Service Reusability and, 280, 380
 - service-oriented analysis processes and, 106
 - Standardized Service Contract and, 147-148, 379
 - support for service capability composition design process, 426
 - Web service region of influence, 370
- service encapsulation**, 235-237, 306
- service enterprise models. *See* service inventory blueprints**
- service granularity**, 486
- cohesion and, 467
 - coupling and, 195-196
 - explained, 116
 - functional context and, 116
 - inheritance (OOAD) and, 473
 - Service Abstraction and, 238-239
 - Service Autonomy and, 311-312
 - Service Composability and, 427-428
 - Service Discoverability and, 378
 - Service Reusability, 277-278
 - Service Statelessness and, 346
 - standardization of service contracts, 142-144

- service instances, 344-346
 - Service Statelessness and, 344-346
- service inventory. *See also* service inventory blueprints
 - analysis process, 521
 - defined, 40
 - delivery processes, 520-521
 - evolutionary stages, 407, 409-410
 - modeling, 520-521
 - example of, 270
 - explained, 40
 - implementation, 42
 - as related to service compositions, 407
 - relationship with service-oriented computing elements, 41
- service inventory blueprints, 53, 313, 320. *See also* service inventory architecture definition, 520
 - case study, 66
 - defining, 520
 - explained, 51-52
 - selecting processes, 521
 - Service Reusability, 269-270
- service inventory models. *See* service inventory blueprints
- service layers, 60, 82
- service level agreement. *See* SLA
- service logic autonomy, 306-308, 488
- Service Loose Coupling (principle), 71, 164-209, 299. *See also* coupling
 - associated terminology, 486
 - association with Service Autonomy principle, 299
 - capability granularity and, 195-196
 - considerations when designing service-oriented classes, 473
 - constraint granularity and, 195-196
 - contribution to realizing organizational agility, 506
 - data granularity and, 195-196
 - effect on other design principles, 197-200
 - interoperability and, 74
 - performance, 202
 - principle profile, 167, 169
 - Service Abstraction principle and, 114, 198, 241
 - Service Autonomy and, 178, 199, 315-316
 - Service Composability and, 199-200, 433
 - Service Discoverability principle and, 199
 - Service Reusability and, 199, 279
 - Standardized Service Contract principle and, 145-146, 173, 198
 - technology abstraction and, 221
 - Web services and, 50
- service methods, explained, 115
- service modeling, 60, 522-525. *See also* service-oriented analysis
 - alternative terms for, 485
 - business analysts and, 53
 - business-centric, 45
 - classification, 485
 - coupling and, 196-197
 - entity services, 44
 - explained, 43-46, 52
 - non-business-centric, 46
 - orchestrated task services, 45
 - process, 523
 - Service Abstraction and, 239
 - Service Autonomy and, 105, 311-314, 525
 - service candidates, 52
 - Service Composability and, 428-430
 - Service Discoverability and, 106, 377-378, 525

- Service Reusability and, 105, 276-278, 525
- Service Statelessness and, 346-347
- service-orientation principles and, 105-106
- service-oriented design processes and, 526-527
- standardization of service contracts, 144
- task services, 44-45
- technology architects and, 53
- utility services, 46
- wrapper service model, 306
- Service Normalization design pattern**, 272, 305, 465, 531
 - service contract autonomy and, 302-304
- service operations, explained, 115
- service policies, standardization of, 137-139
- service profiles, 155
 - capability profiles, structure of, 481-482
 - case study, 155, 157
 - customizing, 482
 - example of, 383-386
 - explained, 478-479
 - policies and, 483
 - principle profiles versus, 110
 - service catalogs and, 483
 - service registries and, 482
 - structure of, 480
- service providers, 48-49
- service registries
 - explained, 366
 - service profiles and, 482
- service registry custodians (role), 493-494
- Service Reusability (principle)**, 62, 72, 254-292, 343, 393, 465, 468
 - agnostic services, 268-269
 - application level terminology, 487
 - in case study, 288-292
 - contribution to realizing organizational agility, 506
 - cultural issues, 281-283
 - design principles, relationship with, 278, 280-281
 - design risks, 281
 - agile delivery*, 287
 - commercial design*, 286-287
 - governance structure*, 283-285
 - organizational culture*, 281-283
 - reliability*, 286
 - security*, 286
- Domain Inventory design pattern** and, 275
- effect on other design principles, 278-281
- explained, 254
- governance issues, 283-285
- interoperability and, 74
- Logic Centralization design pattern**
 - Contract Centralization and*, 272-273
 - difficulty in achieving*, 274-275
 - as enterprise design standard*, 272
 - explained*, 271
 - Web services and*, 274
- measuring, 262-263
 - in analysis/design phase*, 265-266
 - commercial design approach*, 262, 264-265
 - gold-plating*, 267
 - in implementation phase*, 267
- principle profile, 259-261

- reduced IT burden, 64
- Service Abstraction and, 241, 279
- Service Autonomy and, 280, 316
- Service Composability and, 280, 435
- service contracts and, 147
- Service Discoverability and, 280, 380
- service granularity, 277-278
- service inventory blueprints, 269-270
- Service Loose Coupling and, 199, 279
- service modeling and, 105, 276-278, 525
- Service Statelessness and, 280, 348
- service-oriented analysis processes and, 105
- Standardized Service Contract and, 147, 278
- Web services and, 50
- Service Statelessness (principle), 73, 326-359. *See also* state management**
 - in case study, 351-359
 - considerations when designing service-oriented classes, 473
 - contribution to realizing ROI, 505
 - design principles, relationship with, 347-349
 - design risks
 - architecture dependency*, 349-350
 - runtime performance*, 350
 - underestimating effort requirements*, 350
 - effect on other design principles, 347-349
 - explained, 326
 - granularity and, 346
 - interoperability and, 74
 - measuring, 339
 - fully deferred state management*, 342-343
 - internally deferred state management*, 342
 - non-deferred state management*, 340
 - partially deferred memory*, 340-341
 - partially deferred state management*, 341-342
 - messaging as deferral option, 343-344
 - principle profile, 331-332, 334
 - scalability, 261
 - Service Autonomy and, 316, 348
 - Service Composability and, 436
 - service instances and, 344-346
 - service models and, 346-347
 - Service Reusability and, 280, 348
 - state, types of, 335
 - active*, 335
 - business data*, 338
 - context data*, 337-338
 - passive*, 335
 - session data*, 336-337
 - stateful*, 336
 - stateless*, 336
 - state deferral
 - explained*, 329
 - messaging as*, 343-344
 - state delegation versus*, 331
 - state delegation
 - explained*, 329
 - state deferral versus*, 331
 - state management
 - in client-server architectures*, 328
 - databases and*, 329, 331, 339-343
 - in distributed architectures*, 329, 331

- explained*, 327-328
 - origins of*, 328-331
 - performance and*, 334
 - service compositions and*, 340
 - SOAP attachments and*, 334
- service symbol, explained, 13, 15-16
- service-orientation**
 - advantages of, 81-84
 - applications and, 82, 91-92
 - applications versus service compositions, 91-92
 - challenges introduced by, 85-88
 - comparison with object-orientation, 446-475
 - counter-agile delivery and, 87
 - coupling types and, 193-195
 - defined, 39
 - design characteristics, importance of, 69
 - as design paradigm, 30, 70-71
 - design standards and, 86, 89
 - evolution of, 89
 - explained, 68-101
 - governance demands, 88
 - integration and, 84, 92-94
 - interoperability and, 74-75, 84
 - meta abstraction types in, 229-230
 - object-orientation compared, 97, 446-449
 - common goals*, 449-452
 - design principles*, 457-472
 - fundamental concepts*, 453-457
 - origins of, 96-99
 - AOP (aspect-oriented programming)*, 99
 - BPM (business process management)*, 98
 - EAI (enterprise application integration)*, 98-99
 - object-orientation*, 97
 - Web services*, 98
 - problems solved by, 75-84
 - relationship with service-oriented computing elements, 40
 - reusability, level required, 90
 - service compositions, explained, 94-95
 - standardization and, 89
 - technology architectures and, 95-96
 - top-down delivery, 86-87
- service-orientation principles. *See also* design principles
 - service modeling processes and, 105-106
 - service-oriented analysis processes and, 105-106
 - service-oriented design processes and, 106-107
- service-oriented analysis, 60, 522-523. *See also* service modeling
 - business analysts and, 53
 - design principles in, 105-106
 - explained, 52-53
 - process, 521
 - service-orientation principles and, 105-106
 - technology architects and, 53
- service-oriented architecture. *See* SOA *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*, 492
- Service-Oriented Architecture: Concepts, Technology, and Design*, 5, 100, 432, 518
- service-oriented classes, designing, 472-474
- service-oriented computing
 - elements, 37-42
 - explained, 37-54
 - goals and benefits, 55-56

- business and technology domain alignment*, 61
- design principles, relationship with*, 498-499
- increased business and technology domain alignment*, 60-61
- increased federation*, 58-59
- increased intrinsic interoperability*, 56-57
- increased organizational agility*, 63-64
- increased ROI*, 61-62
- increased vendor diversification*, 59
- reduced IT burden*, 64-65
- as related to service-orientation principles*, 104-105
- relationships between*, 56
- vendor diversification*, 59-60
- governance, 88
- implementation, 41-42
- relationships among elements, 40-42
- service compositions and, 39-40
- service inventory and, 40
- service inventory blueprints and, 51-52
- service models and, 43-46
- service-oriented analysis and, 52-53
- service-oriented design and, 53-54
- services and, 39
- SOA and, 38, 56
- terminology, 484-485
- vision, 55
- Web services and, 49-50
- service-oriented design, 377, 521, 525, 527-528
- contract first design, 53, 131, 173, 194
- explained, 53-54
- Service Abstraction
 - design principles, relationship with*, 239-241
 - encapsulation*, 235-237
 - non-technical contract documents*, 237-238
 - service granularity and*, 238-239
 - service models and*, 239
- Service Autonomy
 - design principles, relationship with*, 314-317
 - service granularity and*, 311-312
 - service models and*, 311-314
- Service Composability
 - composition autonomy and*, 430
 - design principles, relationship with*, 432-436
 - orchestration and*, 430, 432
 - service granularity and*, 427-428
 - service models and*, 428-430
- Service Contracts
 - data transformation, avoiding*, 140-142
 - service granularity*, 142-144
 - service models*, 144
- Service Discoverability
 - design principles, relationship with*, 378-380
 - policy assertions and*, 378
 - service granularity and*, 378
 - service models and*, 377-378
- service models and, 526-527
- Service Statelessness
 - design principles, relationship with*, 347-349

- granularity and*, 346
 - messaging as deferral option*, 343-344
 - service instances and*, 344-346
 - service models and*, 346-347
- service-orientation principles and, 106-107
- service-oriented solution logic
 - defined, 39
 - implementation, 42
 - relationship with service-oriented computing elements, 40
- service-to-consumer coupling, 180
- services
 - agnostic, 62, 82, 91
 - business-centric, 45
 - in case study, 154
 - as collections of capabilities, 69-70
 - communications quality, 365
 - as components, 176-177
 - as containers, 70
 - counter-agile delivery of, 87
 - defined, 39
 - dependencies between, 165
 - discoverability, 364-367
 - explained, 39, 68-69
 - as federated endpoints, 58
 - functional context, 70
 - implementation, 42, 47
 - interoperability, 84
 - interpretability, 364-367
 - as IT assets, 62
 - non-business-centric, 46
 - normalized, 65, 83
 - ownership, 88
 - real-world analogy, 68-70
 - relationship with service-oriented computing elements, 40
 - reusable versus agnostic, 268-269
 - reuse. *See* reuse; Service Reusability (principle)
 - ROI, 62
 - roles
 - service consumers*, 48-49
 - service providers*, 48-49
 - scalability, 326, 333, 340, 348
 - service candidates versus, 52
 - standardization of, 89
 - symbols for, 39
 - usage requirements, 318
 - Web services versus, 49
- session data (state management), 336-337
- shared autonomy, 305-306, 488
- silo-based applications, 92
 - advantages of, 76-78
 - counter-federation and, 80
 - disadvantages of, 78-81
 - integration and, 81
 - redundancy, 78
- Simple Object Access Protocol. *See* SOAP
- single responsibility principle. *See* SRP
- single-purpose programs, 255
- SLA (service level agreement), 152-153, 237-238, 249, 382, 386, 483
- SOA (service-oriented architecture), 5. *See also* service-oriented computing
 - explained, 38
 - goals and benefits, 498-499
 - governance, 88
 - relationship with service-oriented computing elements, 40
 - scalability, 326, 333, 340, 348
 - service-oriented computing versus, 56
 - vendor diversified, 60
 - vendor-agnostic, 60

- vision, 55
- Web services and, 46-51
 - architecture*, 48-49
 - standards*, 47-48
- SOA: *Design Patterns*, 4, 31-32, 111, 122, 150, 474, 515, 530-531
- The SOA Magazine Web site, 533
- SOAP (Simple Object Access Protocol), 47
 - attachments, 334, 344
 - headers, 337-338, 344-346, 410
 - processors, 334
- software composition. *See* composition (OOAD)
- specialization (OOAD), 461-462
- SRP (OOAD), 466-468
- standardization. *See also* standards
 - functional expression, 147
 - of service contracts
 - data representation*, 134-137, 140-142, 155
 - design principles, relationship with*, 144-148
 - functional service expression*, 133-134, 155
 - service granularity*, 142-144
 - service models*, 144
 - service policies*, 137-139
 - of vocabularies, 484
- Standardized Service Contract (principle), 71, 464
 - agnostic service contracts and, 144
 - capability granularity, 143
 - case study, 154-161
 - considerations when designing service-oriented classes, 473
 - constraint granularity, 143
 - coupling types, 169-173
 - consumer-to-contract coupling*, 185-191, 214, 473, 486
 - contract-to-functional coupling*, 180
 - contract-to-implementation coupling*, 177-179
 - contract-to-logic coupling*, 174-175
 - contract-to-technology coupling*, 176-177
 - logic-to-contract coupling*, 173-174
 - data granularity, 143
 - design risks, 149
 - development tool deficiencies*, 151-152
 - technology dependencies*, 150
 - versioning*, 149-150
 - design standards and, 132
 - effect on other design principles, 144-148
 - functional meta data, 374
 - origins of, 127-129
 - interoperability and, 74
 - naming conventions, 147
 - non-agnostic service contracts and, 144
 - non-technical service contracts, 152-153
 - principle profile, 130-132
 - Service Abstraction and, 146, 240
 - Service Autonomy and, 301-305, 315
 - Service Composability and, 148, 432
 - Service Discoverability and, 147-148, 379
 - Service Loose Coupling and, 145-146, 173, 198
 - service models and, 144
 - Service Reusability and, 147, 278

- standardization types
 - data representation*, 134-137, 140-142, 155
 - design principles, relationship with*, 144-148
 - functional service expression*, 133-134, 155
 - service granularity*, 142-144
 - service models*, 144
 - service policies*, 137-139
 - transformation and, 140-142
 - Web services and, 50
 - standards. *See also* design standards; standardization
 - SOA, 5-6
 - Web services standards, 47-48
 - www.soaspecs.com Web site, 50
 - state, types of, 335
 - active, 335
 - business data, 338
 - context data, 337-338
 - passive, 335
 - session data, 336-337
 - stateful, 336
 - stateless, 336
 - state data management. *See* state management
 - state databases, 329, 331
 - state deferral
 - explained, 329
 - messaging as, 343-344
 - state delegation versus, 331
 - state delegation
 - explained, 329
 - state deferral versus, 331
 - state management. *See also* Service Statelessness (principle)
 - in client-server architectures, 328
 - databases and, 329, 331, 339-343
 - in distributed architectures, 329, 331
 - explained, 327-328
 - origins of, 328-331
 - performance and, 334
 - service compositions and, 340
 - SOAP attachments and, 334
 - state deferral and state delegation, 329, 331
 - state types, 335
 - active*, 335
 - business data*, 338
 - context data*, 337-338
 - passive*, 335
 - session data*, 336-337
 - stateful*, 336
 - stateless*, 336
 - stateful state (state management), 336
 - stateless state (state management), 336
 - statelessness. *See* Service Statelessness (principle)
 - static business process definition, explained, 397
 - Status (service profile field), 482
 - sub-classes (OOAD), 459, 461, 463
 - sub-controllers, explained, 398, 429
 - super-classes (OOAD), 459
 - symbols
 - color in, 13
 - conflict symbol, 13
 - coupling, 165
 - legend, 13
 - service symbol, 13, 15-16, 39
- T**
- tactical reusability, 487
 - measuring, 265
 - targeted functional coupling, 180
 - targeted reusability, 487
 - measuring, 266

targeted reuse, example of, 289

task services, 340

coupling and, 197

example of, 44

explained, 44-45

functional coupling and, 180

Service Abstraction and, 239

Service Autonomy and, 313-314

service contracts, 144

in service inventory, 270

Service Statelessness and, 347

task-centric business services. *See* task services

technical communications specialists (role), 494

technical service contracts, explained in abstract, 127. *See also* service contracts

technology abstraction

Service Loose Coupling and, 221

Web services and, 221

technology and business alignment. *See* business and technology domain alignment in service-oriented computing

technology architects, role of, 53

technology architecture. *See* architecture

technology coupling, Contract

Centralization design pattern, 189-190

technology dependencies of service contracts, 150

technology information abstraction, 219-221, 225, 486

technology services. *See* utility services

technology transformation, 142

terminology. *See* vocabularies

top-down processes, 86-87, 518-519

traditional solution delivery, explained, 76-81

transformation. *See also* data transformation

avoidance, 135-136, 140-142

design standards and, 135-136

standardization and, 140-142

Standardized Service Contract

principle and, 135-136, 140-142

technology, 142

U

UDDI, 47, 367, 372

UML (unified modeling language), 447, 453

unidirectional coupling, 165

Universal Description, Discovery, and Integration. *See* UDDI

uses-a relationships (OOAD), 469, 471, 474

utility services

coupling and, 197

design processes, 526

explained, 46

Service Abstraction and, 239

Service Autonomy and, 313

service contracts, 144

Service Statelessness and, 347

V

Validation Abstraction design pattern, 531

validation coupling, 190-191

performance and, 202

validation logic

constraint granularity and, 117-118

policies, 137

vendor diversification in service-oriented computing, 59-60

Version (service profile field), 482

versioning, 260, 438
 service contracts and, 149-150
 vocabularies, 147
 for design principle application levels, 487-488
 for design principles, 486-487
 for policy assertions, 137-138
 service contracts, 133
 service models, alternative terms for, 485
 service-oriented computing terminology, 484-485
 standardization of, 484

W

Web Service Contract Design for SOA, 5, 150, 153

Web service regions of influence
 composition members, 395
 designated controllers, 396
 functional abstraction, 225
 programmatic logic abstraction, 226
 quality of service abstraction, 226
 service autonomy, 297
 service contracts, 131-132
 service discoverability, 370
 service loose coupling, 169
 service reusability, 260-261
 service statelessness, 334
 technology information abstraction, 225

Web services, 46-51
 architecture, 48-49
 auto-generation of contracts, 54, 152, 175
 avoiding technology dependencies, 150
 consumer-to-contract coupling and, 186

Contract Centralization design pattern and, 190, 274
 contracts, 134-137
 coupling and, 166
 design processes, 527
 federation via, 59
 first-generation platform, 47
 implementation coupling and, 166
 as implementation medium, 114
 as industry standards, 34
 as influence of service-orientation, 98, 448
 interface element, 456
 Logic Centralization and, 274
 logic-to-contract coupling and, 201
 meta abstraction types and, 225-226, 229-230
 origins of reuse, 258
 origins of Standardized Service Contract principle, 129
 policies. *See* policies; WS-Policy
 portType element, 456
 reuse and, 258
 roles
 service consumers, 48-49
 service providers, 48-49
 Schema Centralization design pattern and, 135-137
 schema custodians (role), 492
 second-generation platform, 47
 service compositions and, 401, 405-406
 service contracts, 127
 service description documents, 127
 service-oriented computing, 49-50
 services versus, 49
 standardization, 134-137
 standards, 47-48
 technology abstraction and, 221

- technology-to-contract coupling and, 177
 - validation coupling and, 190-191
- Web Services Business Process Execution Language. *See* WS-BPEL
- Web Services Description Language. *See* WSDL
- Web services tutorials Web site, 50, 534
- Web sites
 - www.soabooks.com, 16, 531, 533
 - www.soaglossary.com, 16, 533
 - www.soamag.com, 17, 533
 - www.soaposters.com, 16, 534
 - www.soaspecs.com, 16, 338, 460, 493, 533
 - www.thomaserl.com, 17
 - www.ws-standards.com, 338, 432, 534
 - www.xmlenterprise.com, 534
- wrapper services, 306, 316
 - Service Autonomy design risks, 318
- WS-* extensions, 47, 395
- WS-Addressing, 337-338, 344-346, 373
- WS-AtomicTransaction, 338
- WS-BPEL, 197, 239, 431-432, 527
- WS-Coordination, 338
- WS-I Basic Profile, 47, 150-151
- WS-MetadataExchange, 372-373
- WS-Policy, 48, 129, 131, 483, 493
- WS-Policy definitions, 127, 137-139, 146, 151, 153, 274
 - contract-to-logic coupling, 179
 - editors, 152
 - structural standards, 139
 - wsp:optional attribute, 139
- WS-ReliableMessaging, 137
- WS-ResourceTransfer (WS-RT), 338
- WS-SecurityPolicy, 137
- WSDL (Web Services Description Language), 47, 129, 131, 146, 174, 274
- WSDL definitions, 127, 175
 - auto-generation, 175
 - standardization, 136
 - XML schemas and, 136
- wsp:ignorable attribute, 143, 238, 378
- wsp:optional attribute, 139, 143, 238, 378
- X–Z**
- XML, 194
 - as industry standards, 34
 - parsers, 334
- XML Schema Definition Language, 47, 129, 131
- XML schemas, 127, 137, 146, 174-175, 274, 455
 - case study, 157
 - constraint granularity example, 117
 - entity schemas, 136
 - schema custodians (role), 492
 - standardization, 136
 - validation coupling and, 190-191
 - WSDL definitions and, 136
- XML tutorials Web site, 534
- XSD. *See* XML Schema Definition Language; XML schemas
- XSLT, 140