THE PRENTICE HALL SERVICE TECHNOLOGY SERIES FROM THOMAS ERL

SECOND EDITION

Service-Oriented Architecture

Analysis and Design for Services and Microservices

PRENTICE HALL

ServiceTech PRESS

Thomas Erl

About This E-Book

EPUB is an open, industry-standard format for e-books. However, support for EPUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer’s Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the e-book in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

Service-Oriented Architecture

Analysis and Design for Services and Microservices

Thomas Erl

With contributions by Paulo Merson and Roger Stoffers

BOSTON • COLUMBUS • INDIANAPOLIS • NEW YORK • SAN FRANCISCO
AMSTERDAM • CAPE TOWN • DUBAI • LONDON • MADRID • MILAN • MUNICH
PARIS • MONTREAL • TORONTO • DELHI • MEXICO CITY • SAO PAULO
SIDNEY • HONG KONG • SEOUL • SINGAPORE • TAIPEI • TOKYO

Library of Congress Control Number: 2016952031

ISBN-13: 978-0-13-385858-7
ISBN-10: 0-13-385858-8

First printing: December 2016

Publisher

Mark Taub

Editor-in-Chief
Greg Wiegand **Senior Acquisitions Editor**
Trina MacDonald **Managing Editor**
Sandra Schroeder **Senior Project Editors**
Lori Lyons
Betsy Gratner **Copyeditors**
Paula Lowell
Language Logistics
Infinet Creative Group
Maria Lee
Teejay Keepence **Indexer**
Cheryl Lenser

Proofreaders
Williams Woods Publishing
Abigail Gavin
Melissa Mok
Kam Chiu Mok
Shivapriya Nagaraj
Catherine Shaffer
Pamela Janice Yau
Maria Lee **Editorial Assistant**
Olivia Basegio **Cover Design**
Thomas Erl

Photos
Thomas Erl

Cover Compositor
Chuti Prasertsith **Compositor**
Bumpy Design

Graphics
Jasper Paladino
Zuzana Cappova
Infinite Creative Group
Spencer Fruhling
Tami Young
Demian Richardson

Kan Kwai Lui
Briana Lee **Educational Content Development**
Arcitura Education Inc.

To Markus, who recently joined our team with a keen sense of curiosity and a relentless desire to analyze and redesign even the most micro of things.

—Thomas Erl

Contents at a Glance

Contents

Index

Acknowledgments

This Second Edition is comprised of content from a variety of sources, including new content that reflects industry developments and revised content from other series titles. Thank you to all who helped shape what this book is comprised of, and special thanks to the following individuals who contributed new insights and new design patterns: In alphabetical order: • Paulo Merson

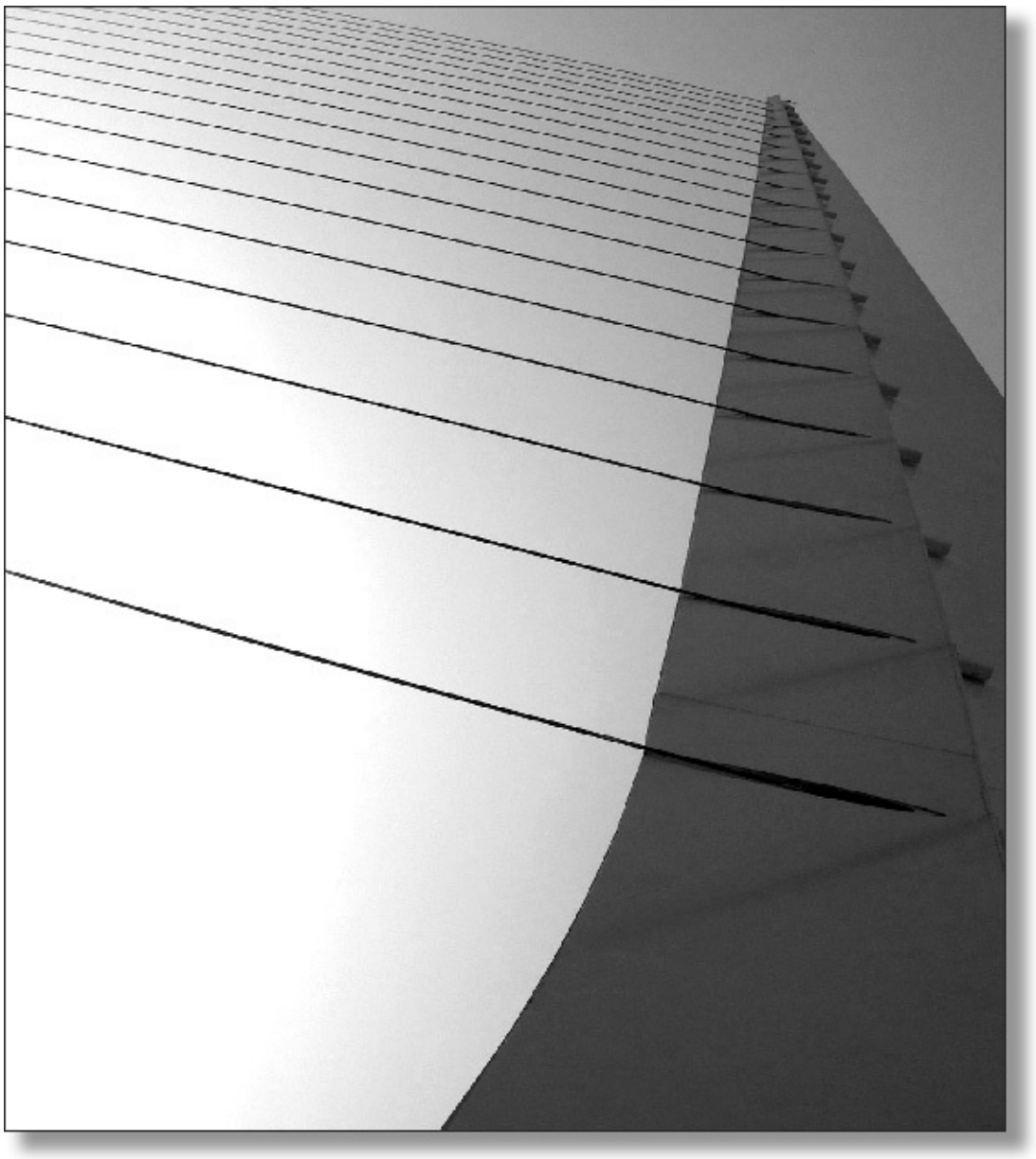
• Roger Stoffers

Reader Services

Register your copy of *Service-Oriented Architecture: Analysis and Design for Services and Microservices* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account.* Enter the product ISBN, 9780133858587, and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”

*Be sure to check the box that you would like to hear from us in order to receive exclusive discounts on future editions of this product.

Chapter 1. Introduction



When I first authored *ServiceOriented Architecture: Concepts, Technology, and*

Design back in 2004, I did so primarily out of a motivation to help organize what at the time was a fragmented whirlwind of misperceptions, ambiguities, and bits of actual valid knowledge about what SOA was and was expected to become. The goal was to establish essential coverage of its architectural model and its underlying design paradigm, along with documentation of the methodology and technology required to achieve it.

I am still humbled by the success this book has had for more than a dozen years. When I was asked to put together a second edition, it seemed like a naturally sound idea. However, when I got down to working on this project it became clear that the scope of this new edition would have to be significantly different from the original title.

Since *ServiceOriented Architecture: Concepts, Technology, and Design* was published, I have authored or co-authored 11 additional books as part of the *Prentice Hall Service Technology Series from Thomas Erl*, eight of which were dedicated to SOA. Each of these eight subsequently released titles expanded upon topics first covered in *ServiceOriented Architecture: Concepts, Technology, and Design.*

This made me think carefully about what should and should not be part of this second edition. Revisiting topics pertaining to technology did not make sense because they had been covered exhaustively in several other titles. However, some of the subsequently released books provide coverage of architecture, design, and methodology that is more current and comprehensive than what was originally documented in the first edition of *ServiceOriented Architecture: Concepts, Technology, and Design*. Repurposing and compiling this content as part of this second edition so that the original scope and purpose of the book could be preserved did make sense, while benefiting from the decade or so of research and authoring that occurred since the publication of the first edition.

This compiled content primarily comprises the three chapters in Part I of this book together with new content pertaining to the formal introduction of microservices to SOA. The chapters in Part II focus solely on serviceoriented analysis and design with some updates and new content pertaining to REST services and microservices.

Specifically, content from the following additional books has been repurposed, revised, and/or incorporated into this second edition:

- *SOA Principles of Service Design*
- *SOA Design Patterns*
- *SOA with REST: Principles, Patterns & Constraints for Building*

Enterprise Solutions with REST

- *Next Generation SOA: A Concise Introduction to Service Technology & ServiceOrientation*
- *SOA Governance: Governing Shared Services On-Premise & in the Cloud*

Select content has been updated, and some of it has been further augmented to incorporate the microservice model and micro task service layer.

I hope you find value in what's been put together. It's genuinely the best possible second edition of the original title that could be assembled. The fact that this second edition looks so much different from the first is a tribute to the tremendous progress that has been made in the evolution and maturation of modern-day, serviceoriented architecture.

1.1 How Patterns Are Used in this Book

When the first edition of *ServiceOriented Architecture: Concepts, Technology, and Design* was authored, we had not yet embarked on the creation of what was to become the SOA design patterns catalog. Since the initial version of the patterns catalog was published in 2008 at www.soapatterns.org, it has steadily grown and accompanying, complementary pattern catalogs have emerged for cloud computing (www.cloudpatterns.org) and Big Data (www.bigdatapatterns.org).

Patterns have also become an important part of the language used to author books in this series. Most books published since the release of the SOA patterns catalog contain references to relevant patterns, and some even introduce new ones.

Because this is the second edition of a book that originally did not contain patterns, it was written without any requirement to know or understand patterns. Instead, wherever appropriate, *SOA Patterns* sections have been inserted. These sections highlight patterns relevant to preceding content. Appendix C contains summarized profiles of all referenced patterns. Inline page references are used to link pattern references with profiles, as explained in the upcoming *Page References and Capitalization for Principles, Constraints, and Patterns* section.

So, even though patterns do not need to be understood or studied to complete this book, it is highly recommended that you take the time to do so anyway. If you are brand new to the world of design patterns, be sure to read the introductory section at the beginning of Appendix C or the more comprehensive tutorial in Chapter 5 of the *SOA Design Patterns* book.

1.2 Series Books That Cover Topics from the First Edition

As mentioned earlier, a number of topics from the first edition of this book were subsequently covered more comprehensively in subsequent titles released as part of the *Prentice Hall Service Technology Series from Thomas Erl*.

For those of you familiar with the first edition, let's revisit the original chapters so that we can identify those that remained part of this second edition and then map the others to series titles that elaborated on their respective topic areas.

- *Chapter 2, Case Studies* – This chapter in the second edition contains abbreviated case study backgrounds from the first edition of *ServiceOriented Architecture: Concepts, Technology & Design* and *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*.
- *Chapter 3, Introducing SOA* – The topics in this chapter have been significantly updated with content from Chapter 3 of *SOA Principles of Service Design* and Chapter 4 of *SOA Design Patterns*.
- *Chapter 4, The Evolution of SOA* – Chapter 4 of *SOA Principles of Service Design* covers historical origins of serviceorientation and Chapters 3 and 4 of *SOA Design Patterns* contrasts SOA to other architectural models.
- *Chapter 5, Web Services and Primitive SOA*, *Chapter 6, Web Services and Contemporary SOA Part I*, and *Chapter 7, Web Services and Contemporary SOA Part II* – Web service technologies and markup languages are covered in detail in *Web Service Contract Design and Versioning for SOA*.
- *Chapter 8, Principles of ServiceOrientation – SOA Principles of Service Design* is dedicated to documenting the eight serviceorientation principles. Chapter 3 in this second edition provides more detailed coverage of serviceorientation with sections that originated in *SOA Principles of Service Design*.
- *Chapter 9, Service Layers* – Chapters 6 and 7 of *SOA Design Patterns* provide a series of design patterns that formally document established service layers. Service layers are covered in Chapter 5 of this second edition and the new micro task service layer is introduced.
- *Chapter 10, SOA Delivery Strategies* – Chapter 5 of *SOA Governance: Governing Shared Services On-Premise & in the Cloud* covers project stages and Chapter 6 addresses methodology. The end of Chapter 4 in this second edition summarizes project stages and related organizational roles.

- *Chapter 11, ServiceOriented Analysis Part I* and *Chapter 12, ServiceOriented Analysis Part II* – Topics from these chapters are revisited in Chapters 6 and 7 of this second edition, and are further supplemented with updated analysis content from *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*.
- *Chapter 13, ServiceOriented Design Part I* and *Chapter 14, ServiceOriented Design Part II* – The markup languages from this chapter are covered in more detail in the *Web Service Contract Design and Versioning for SOA* book.
- *Chapter 15, ServiceOriented Design (Part III, Service Design)* – Topics from this chapter are revisited in Chapters 8 and 9 of this second edition and are further supplemented with updated design content from *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*.
- *Chapter 16, ServiceOriented Design (Part IV, Business Process Design)* – Coverage of orchestration-related technologies is provided in various sections in *SOA with .NET: Realizing ServiceOrientation with the Microsoft Platform* and *SOA with Java: Realizing ServiceOrientation with Java Technologies*.
- *Chapter 17, Fundamental WS-* Extensions* – Several of the standards from this chapter are covered in more detail in *Web Service Contract Design and Versioning for SOA*.
- *Chapter 18, SOA Platforms* – The documentation of SOA support in .NET and Java platforms is provided comprehensively in the corresponding *SOA with .NET: Realizing ServiceOrientation with the Microsoft Platform* and *SOA with Java: Realizing ServiceOrientation with Java Technologies* titles.

For more information about any of the aforementioned books from the *Prentice Hall Service Technology Series from Thomas Erl*, visit www.servicetechbooks.com.

1.3 How this Book Is Organized

This book begins with Chapters 1 and 2, which supply introductory content and case study background information, respectively. Provided here is a brief overview of subsequent chapters.

Part I: Fundamentals

Chapter 3: Understanding ServiceOrientation

This chapter provides detailed coverage of the serviceorientation design paradigm, including its underlying design philosophy and design principles, as well as a comparison to traditional silo-based design approaches. The chapter concludes with coverage of typical critical success factors for adopting serviceorientation within organizations.

Chapter 4: Understanding SOA

This chapter delves into the distinct characteristics and types of serviceoriented architecture and further explores the links between the application of the serviceorientation design paradigm and technology architecture. The chapter concludes with brief coverage of common SOA project lifecycle stages and organizational roles, with an emphasis on the service inventory analysis, serviceoriented analysis, and serviceoriented design phases.

Chapter 5: Understanding Layers with Services and Microservices

This chapter provides an updated version of the standard service models and corresponding service layers. It incorporates this new content into a new service definition process with the addition of the microservice model and micro task service layer. The relevance of service deployment bundles and containerization are also briefly mentioned in relation to microservice implementation requirements.

Part II: ServiceOriented Analysis and Design

Chapter 6: Analysis and Modeling with Web Services and Microservices

Updated, step-by-step coverage of the serviceoriented analysis process for Web services, along with case study examples. Microservice identification as part of a Web services analysis is covered, but microservice modeling is deferred to Chapter 7.

Chapter 7: Analysis and Modeling with REST Services and Microservices

The serviceoriented analysis process for REST-based services is revised with the incorporation of microservices. This chapter is also supplemented with updated case study examples.

Chapter 8: Service API and Contract Design with Web Services

Guidelines and service contract design considerations for Web services, along

with an extended case study example.

Chapter 9: Service API and Contract Design with REST Services and Microservices

Service model-specific REST contract design considerations are revised to include microservices. Design guidelines are provided, along with a section dedicated to complex method design. Revised case study examples are also provided.

Chapter 10: Service API and Contract Versioning with Web Services and REST Services

This chapter contains a series of fundamental versioning techniques and considerations for Web service and REST service contracts and APIs.

Part III: Appendices

Appendix A: ServiceOrientation Principles Reference

This appendix provides the profile tables (originally from *SOA Principles of Service Design*) for the serviceorientation design principles referenced in this book.

Appendix B: REST Constraints Reference

This appendix provides the profile tables for the REST design constraints referenced in this book (originally from *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*).

Appendix C: SOA Design Patterns Reference

This appendix provides the profile tables for the SOA design patterns referenced in this book (originally from *SOA Design Patterns* and www.soapatterns.org).

Appendix D: The Annotated SOA Manifesto

This appendix contains the complete annotated version of the SOA Manifesto (originally from *Next Generation SOA: A Concise Introduction to Service Technology & ServiceOrientation* and www.soa-manifesto.com).

1.4 Page References and Capitalization for Principles, Constraints, and Patterns

Each design constraint, principle, and pattern discussed in this book has a

corresponding profile. A *profile* is a concise definition that summarizes key design aspects and considerations. A primary and ongoing topic area of this book is the exploration of how constraints, principles, and patterns relate to and affect each other. You are therefore encouraged to repeatedly refer to the profiles whenever encountering a constraint, principle, or pattern in a context that is unclear to you.

To facilitate the quick reference of profiles, a special convention is used. Each principle, pattern, and constraint name is always capitalized and followed by a page number that points to the corresponding profile page. This convention was established by the design patterns community and is further being extended to design principles and design constraints in this book.

All page references point to profile tables located in the appendices. The profile tables for constraints are provided in Appendix B, and those for principles and patterns are located in Appendices A and C, respectively.

To maintain an immediately recognizable distinction between constraints, principles, and patterns throughout this book, each uses a different delimiter for page numbers. The page number for each constraint is displayed in curly braces, for each principle it is placed in rounded parentheses, and for patterns, square brackets are used, as follows:

- Principle Name (*page number*)
- Constraint Name {page number}
- Pattern Name [*page number*]

For example, the following statement first references a serviceorientation design principle, then an SOA design pattern, and finally a REST constraint:

"…the Service Loose Coupling (293) principle is supported via the application of the Decoupled Contract [337] pattern and the Stateless {308} constraint ..."

In this statement, each reference is explicitly qualified as a principle, pattern, or constraint. Most of the references in this book (especially in later chapters) omit this qualifier to allow for more concise content.

For example, the preceding statement will more commonly be worded as follows:

"…Service Loose Coupling (293) is supported via the application of Decoupled Contract [337] and Stateless {308}…"

This wording convention also has origins within the design patterns community. As previously stated, if you run into a reference without an explicit qualifier, use the page number delimiter (parentheses, square brackets, or curly braces) to

identify its type (principle, pattern, or constraint).

Additional Information

The following sections provide supplementary information and resources for the *Prentice Hall Service Technology Series from Thomas Erl*.

Symbol Legend

This book contains a series of diagrams that are referred to as *figures*. The primary symbols used throughout all figures are described in a symbol legend you can download from www.arcitura.com/notation.

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

You can find information about other series titles and various supporting resources at www.servicetechbooks.com. You are encouraged to visit this site regularly to check for content changes and corrections.

ServiceOrientation (www.serviceorientation.com)

This site provides papers, book excerpts, and various content dedicated to describing and defining the serviceorientation paradigm, associated principles, and the serviceoriented technology architectural model.

What Is REST? (www.whatisrest.com)

This website contains excerpts from this book and related content to provide a concise overview of REST architecture and constraints.

Referenced Specifications (www.servicetechspecs.com)

The chapters throughout this book reference various industry specifications and standards. The www.servicetechspecs.com website provides a central portal to the original specification documents created and maintained by the primary standards organizations.

SOASchool.com® SOA Certified Professional (SOACP)

The SOA Certified Professional curriculum from Arcitura Education is dedicated to specialized areas of serviceoriented architecture and serviceorientation, including analysis, architecture, governance, security, .NET development, Java development, and quality assurance.

For more information, visit www.soaschool.com.

CloudSchool.com™ Cloud Certified Professional (CCP)

The Cloud Certified Professional curriculum from Arcitura Education is dedicated to specialized areas of cloud computing, including technology, architecture, governance, security, and storage.

For more information, visit www.cloudschool.com.

BigDataScienceSchool.com™ Big Data Science Certified Professional (BDSCP)

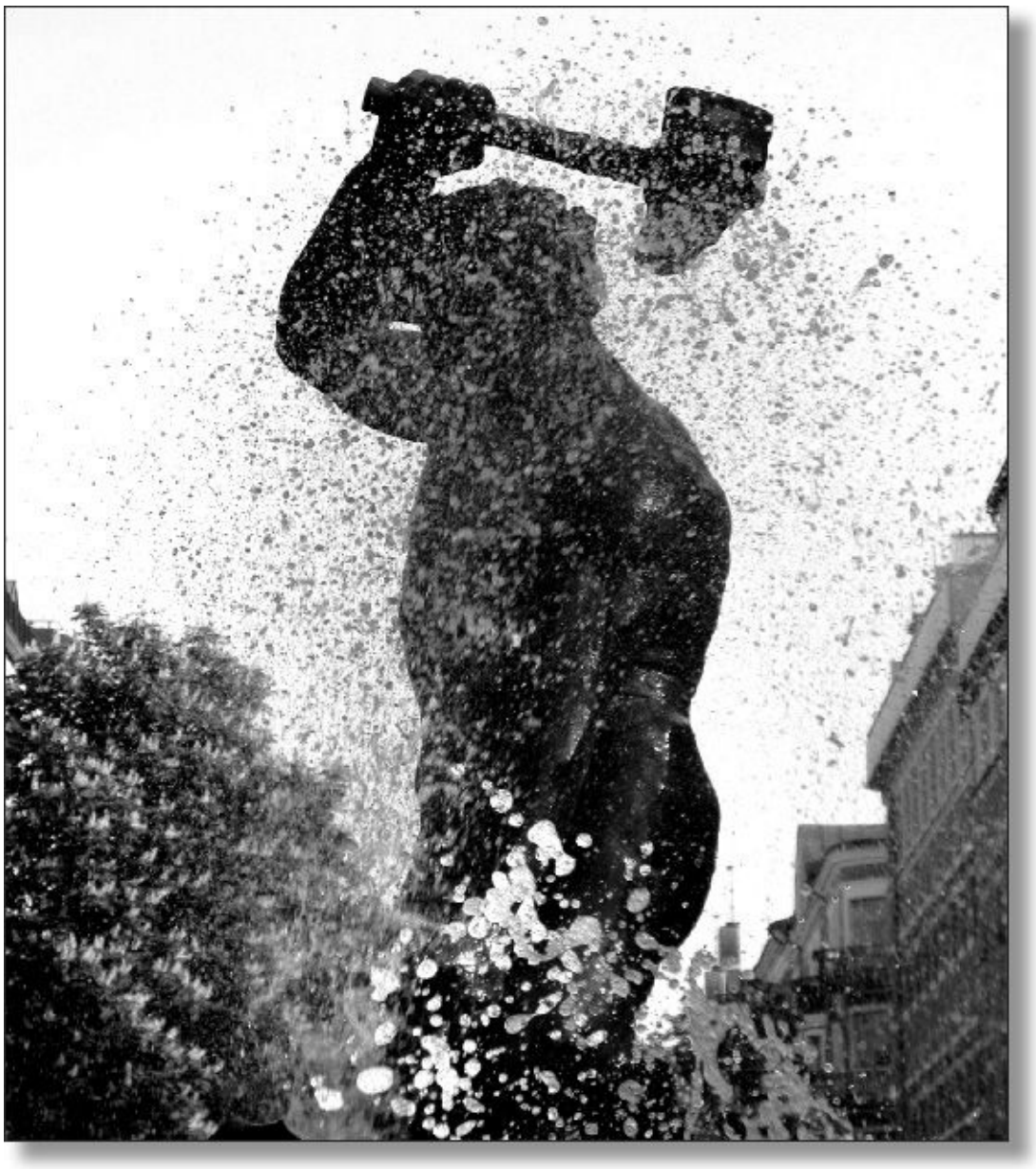
The Big Data Science Certified Professional curriculum from Arcitura Education is dedicated to specialized areas of Big Data analysis and technology, including analytics, engineering, architecture, and governance.

For more information, visit www.bigdatascienceschool.com.

Notification Service

If you would like to be automatically notified of new book releases in this series, new supplementary content for this title, or key changes to the previously listed websites, use the notification form at www.servicetechbooks.com.

Chapter 2. Case Study Backgrounds



2.1 How Case Studies Are Used

Case study examples are an effective means of exploring abstract topics within

real-world scenarios. The information provided in this brief chapter establishes the basis for two separate storylines that relate to *Case Study Example* sections in Chapters 6 to 9. To help you more easily identify these sections, a light gray background is used.

Background information is provided for two different organizations. The first is Transit Line Systems, Inc. (TLS), a private sector corporation. The other is Midwest University Association (MUA), a public sector academic institution.

2.2 Case Study Background #1: Transit Line Systems, Inc.

Transit Line Systems, Inc. (TLS) is a prominent corporation in the private transit sector. It employs more than 1,800 people and has offices in four cities. Although its primary line of business is providing private transit, it has a number of secondary business areas that include a maintenance and repair branch that outsources TLS service technicians to public transit sectors, and a tourism branch that partners with airlines and hotels. Of the 200 IT professionals who support TLS's automation solutions, approximately 50% are contractors who are hired on a per-project basis.

TLS is a corporation that has undergone a great deal of change over the past decade. The identity and structure of the company has been altered numerous times, mostly because of corporate acquisitions and the subsequent integration processes. Its IT department has had to deal with a volatile business model and regular additions to its supported set of technologies and automation solutions. TLS's technical environment therefore is riddled with custom-developed applications and third-party products that were never intended to work together.

The cost of business automation has skyrocketed, as the effort required to integrate these many systems has become increasingly complex and onerous. Not only has the maintenance of automation solutions become unreasonably expensive, but their complexity and lack of flexibility have significantly slowed down the IT department's ability to respond to business change.

Tired of having to continually invest in a non-functional technical environment, IT directors decide to adopt SOA as the standard architecture to be used for new applications. Web services are chosen as the primary technology-set to federate existing legacy systems. The driving motivation behind this decision is a desperate need to introduce enterprise-wide standardization and increase organizational agility.

2.3 Case Study Background #2: Midwest University Association

Midwest University Association (MUA) is one of the oldest educational institutions west of the Mississippi in the continental U.S. It's rated among the top 10 leading universities in the engineering and research fields, and has six remote locations along with its main campus that employ more than 6,000 faculty and staff.

Each program within the university has an independent IT staff and budget to support systems management. The remote campuses also have their own IT departments. Collaboration with external educational institutions is governed by an independent, central enterprise architecture group.

There are various automated solutions for common processes, such as student enrollment, course cataloging, accounting, financials, as well as grading and reporting. The primary system for record keeping is an IBM mainframe that is reconciled every night with a batch feed from the individual remote locations. The different schools themselves employ a variety of technologies and platforms.

After a careful assessment of the existing infrastructure, it is decided to re-engineer several IT systems to a service-oriented architecture that will preserve legacy assets, simplify integration between various internal and external systems, and improve channel experience for both the students and staff. The enterprise architecture group at MUA has proposed a phased adoption of SOA via the use of REST services that can be leveraged across schools and from remote locations.

Part I: Fundamentals

Chapter 3. Understanding Service-Orientation

This chapter is dedicated to describing the service-orientation design paradigm,

its principles, and how it compares to other design approaches.

3.1 Introduction to Service-Orientation

In the everyday 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. Any group of individuals collectively performing a task in support of a larger task is also demonstrating the delivery of a service (Figure 3.1).

Figure 3.1 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 3.2).

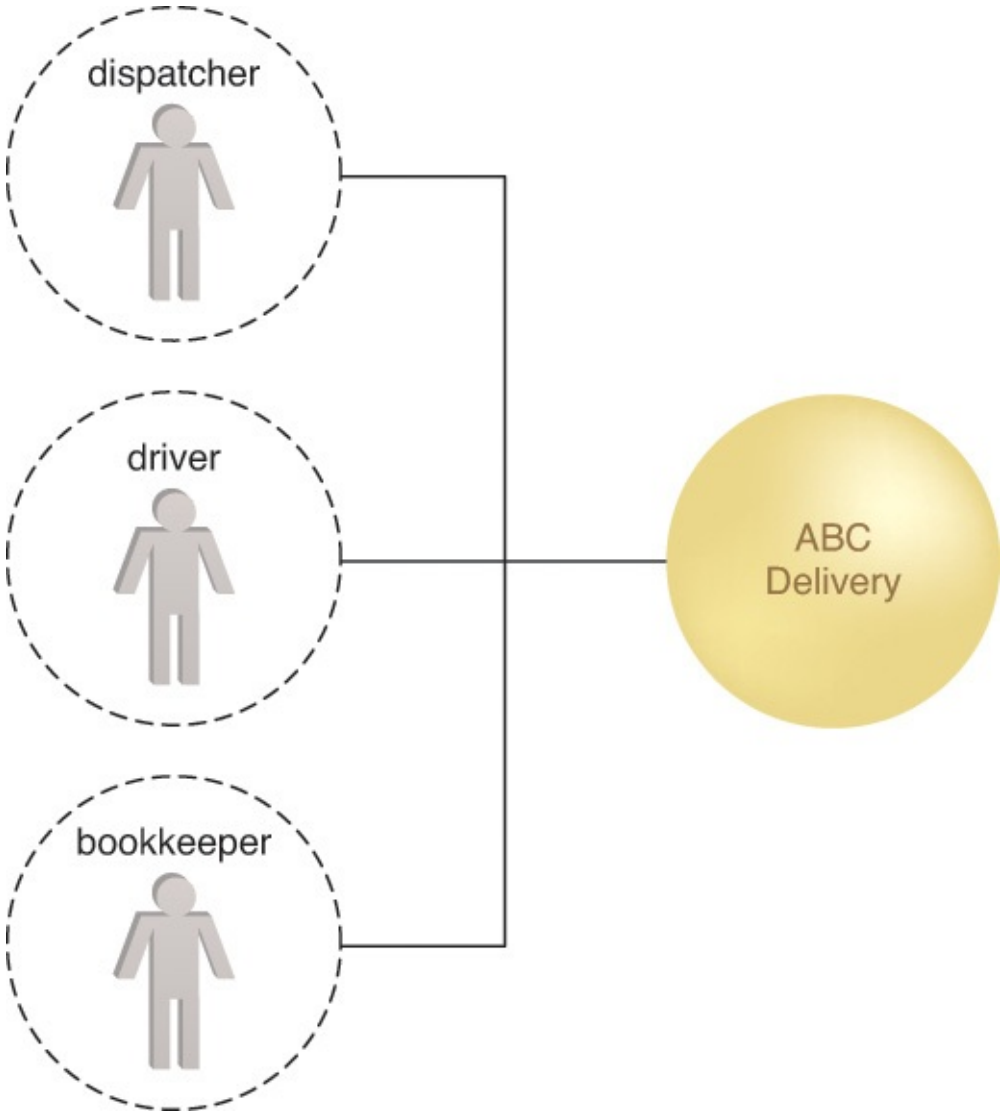


Figure 3.2 A company that employs these three people can compose their capabilities to carry out its business.

Certain baseline requirements exist to enable a group of individual service providers to collaborate in order to collectively provide a larger service. Figure 3.2, for example, displays a group of employees who 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 these things in place, these individuals can be composed into a productive working team. Establishing these types of baseline requirements within and across business automation solutions is a key goal of service-orientation.

Services in Business Automation

From a general perspective, a *service* is a software program that makes its functionality available via a published API that is part of a *service contract*. Figure 3.3 shows the symbol used to depict a service (without providing any detail regarding its service contract).

Figure 3.3 The symbol used to represent an abstract service.

Different implementation technologies can be used to program and build services. The two common implementation mediums covered in this book are SOAP-based Web services (or just Web services) and RESTful services (or just REST services). Figure 3.4 shows the standard symbols used to represent service contracts in this book.

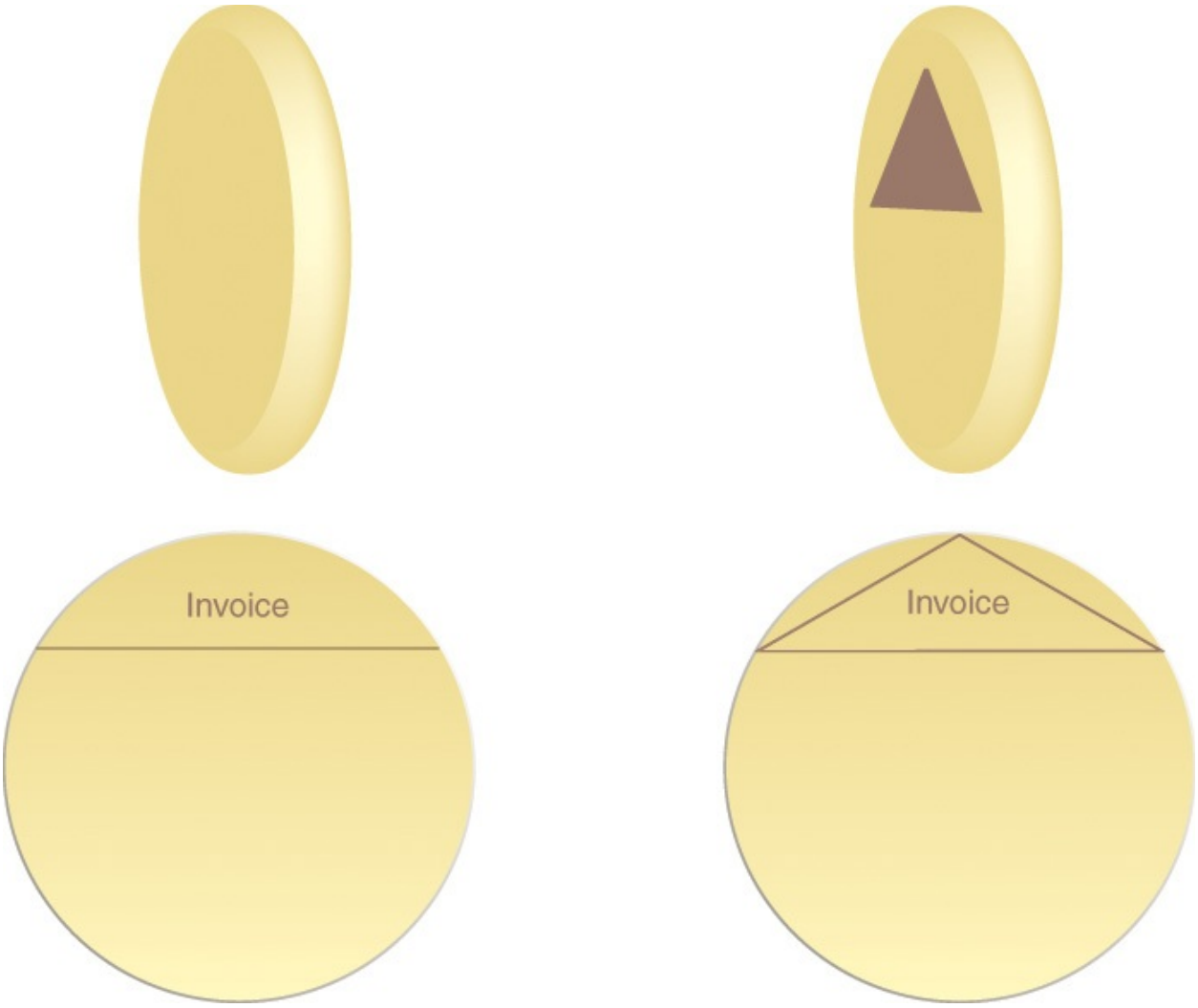


Figure 3.4 The chorded circle symbol used to display an Invoice service contract (left), and a variation of this symbol used specifically for REST service contracts (right).

Note

A Web service contract is generally comprised of a WSDL definition and one or more XML Schema definitions. Services implemented as REST services are accessed via a uniform contract, such as the one provided by HTTP and Web media types. Chapters 8 and 9 provide examples of Web service and REST service contracts.

A service contract can be further comprised of human-readable documents, such as a Service Level Agreement (SLA) that describes additional quality-of-service guarantees, behaviors, and limitations.

Several SLA-related requirements can also be expressed in machine-readable formats.

Services Are Collections of Capabilities

When discussing services, it is important to remember that a single service can offer an API that provides 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 3.5, for example, is that of "shipment." This particular service provides a set of capabilities associated with the processing of shipments.

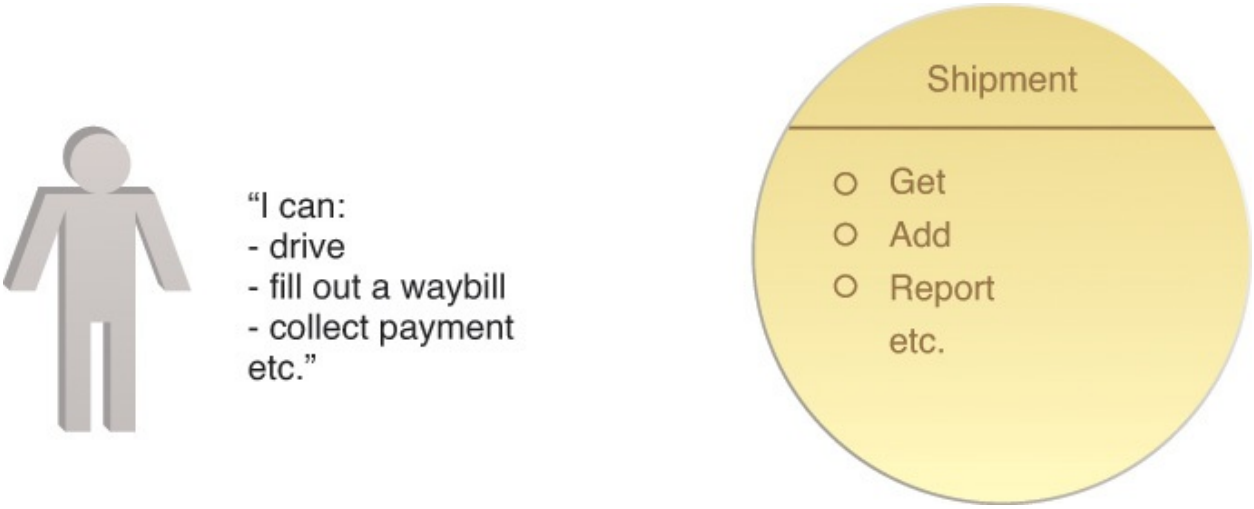


Figure 3.5 Much like a human, an automated service can provide multiple capabilities.

A service is therefore essentially 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. When we make reference to service capabilities in this book, we are specifically focused on those that are defined as part of the service contract API.

A *service consumer* is the runtime role assumed by a software program when it accesses and invokes a service—or, more specifically, when it sends a message to a service capability expressed in the service contract. Upon receiving the request, the service begins executing logic encompassed by the invoked capability and it may or may not return a corresponding response message to the service consumer. A service consumer can be any software program capable of invoking a service via its API. A service itself may act as the consumer of

another service.

Agnostic vs. Non-Agnostic Logic

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

Another way of conceptualizing agnostic and non-agnostic logic is to focus on the extent to which the logic can be repurposed. Due to the multipurpose nature of agnostic logic, it is expected to become reusable across different contexts so that the logic, as a single software program (or service), can be used to help automate multiple business processes. Non-agnostic logic is not subject to these types of expectations. It is deliberately designed as a single-purpose software program (or service) and therefore has different characteristics and requirements. Non-agnostic logic can still be reusable, but only within the scope of its parent business process, which preserves its context as being specific to a greater, single-purpose task.

Service-Orientation as a Design Paradigm

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.

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 with specific characteristics and in support of a specific target state.

Fundamentally, service-orientation shapes suitable units of solution logic as

enterprise resources that can be designed to solve immediate concerns while still remaining agnostic to the greater problem. This provides the constant opportunity to reutilize the capabilities within those units to solve other problems as well.

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." (Chapter 5 explores in detail how the separation of concerns is carried out with service-orientation.)

Services, as part of service-oriented solutions, exist as physically independent software programs with distinct design characteristics. Each service is assigned its own distinct functional context and is comprised of a set of capabilities related to this context. A *service composition* is a coordinated aggregate of services. As explained later in the *Effects of Service-Orientation on the Enterprise* section, a composition of services (Figure 3.6) is comparable to a traditional application in that its functional scope is usually associated with the automation of a parent business process.

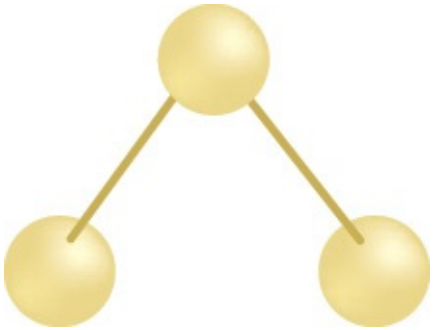


Figure 3.6 This symbol, comprised of three connected spheres, represents a service composition. Other, more detailed representations are based on the use of chorded circle symbols that illustrate which service capabilities are actually being composed.

A *service inventory* is an independently standardized and governed collection of complementary services within a boundary that represents an enterprise or a meaningful segment of an enterprise. Figure 3.7 establishes the symbol used to represent a service inventory in this book.

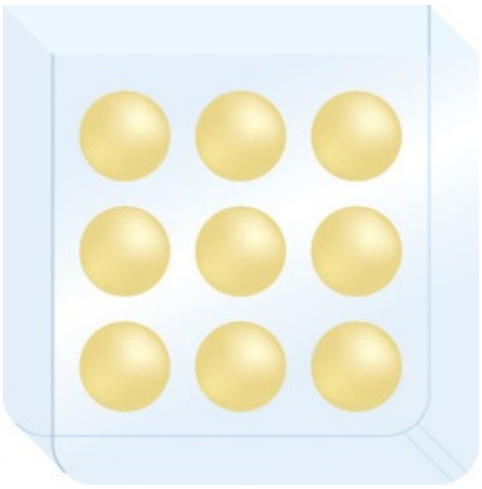


Figure 3.7 The service inventory symbol is comprised of spheres within a container.

An IT enterprise can contain or may even be comprised of a single service inventory. Alternatively, an enterprise environment can contain multiple service inventories. When an organization has multiple service inventories, this term is further qualified as *domain service inventory*.

The application of service-orientation throughout a service inventory is of paramount importance to establish a high degree of native interservice interoperability. This supports the repeated creation of effective service compositions (Figure 3.8).

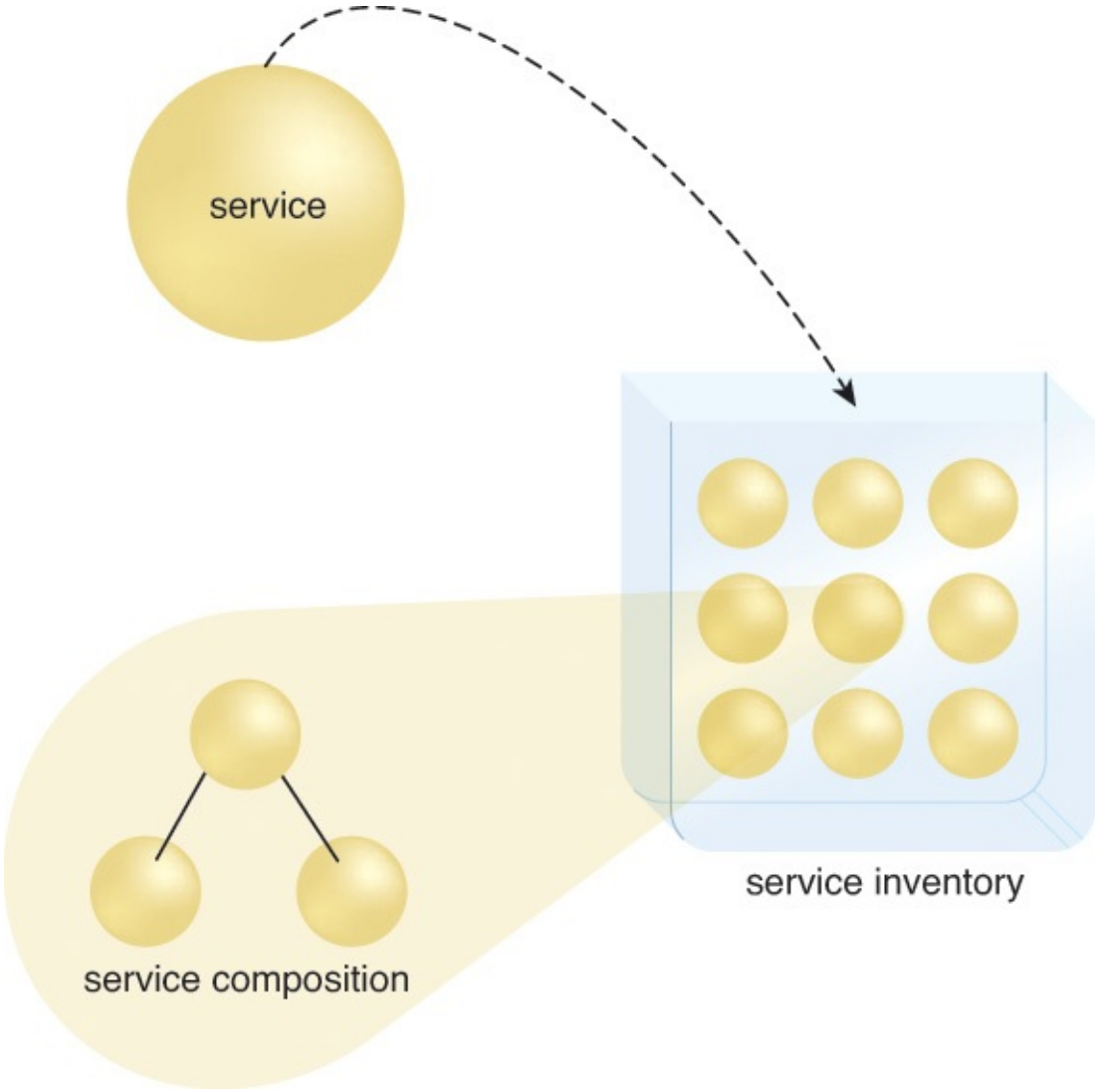


Figure 3.8 Services (top) are delivered into a service inventory (right) from which service compositions (bottom) are drawn.

Here's a brief recap of the elements of service-orientation that have been covered so far:

- *Service-oriented solution logic* is implemented as *services* and *service compositions* designed in accordance with *service-orientation*.
- A *service composition* is comprised of *services* that have been assembled to provide the functionality required to automate a specific business task or process.
- Because *service-orientation* shapes many *services* as enterprise resources, one *service* may be invoked by multiple consumer programs, each of which can involve that same *service* in a different *service composition*.
- A collection of standardized *services* can form the basis of a *service*

inventory that can be independently governed within its own physical deployment environment.

- Multiple business processes can be automated by the creation of *service compositions* that draw from a pool of existing agnostic *services* that reside within a *service inventory*.

As explored in Chapter 4, service-oriented architecture is a form of technology architecture optimized in support of services, service compositions, and service inventories.

Service-Orientation Design Principles

The preceding sections have described the service-orientation paradigm at a very high level. But how exactly is this paradigm applied? It is primarily applied at the service level (Figure 3.9) via the application of the following eight design principles:

- **Standardized Service Contract (291)** – *Services within the same service inventory are in compliance with the same contract design standards.*

  Services express their purpose and capabilities via a service contract. This is perhaps the most fundamental principle in that it essentially dictates the need for service-oriented solution logic to be partitioned and distributed in a standardized manner. It also places a great deal of emphasis on the design of service contracts to ensure that the manner in which services express functionality and define data types is kept in relative alignment.

- **Service Loose Coupling (293)** – *Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment.*

  Coupling refers to a measure of dependency between two things. This principle establishes a specific type of relationship within and outside of service boundaries, with a constant emphasis on reducing ("loosening") dependencies between a service contract, its implementation, and service consumers. Service Loose Coupling (293) promotes the independent design and evolution of service logic while still guaranteeing baseline interoperability.

- **Service Abstraction (294)** – *Service contracts only contain essential information and information about services is limited to what is published in service contracts.*

  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 the previously described loosely coupled relationship. Service Abstraction (294) also plays a significant role in the positioning and design of service compositions.

- **Service Reusability (295)** – *Services contain and express agnostic logic and can be positioned as reusable enterprise resources.*

  Whenever we build a service, we look for ways to make its underlying capabilities useful for more than just one purpose. Reuse is greatly emphasized with service-orientation—so much so, that it becomes a core part of the design process and it also forms the basis for key service models (as explained in Chapter 5).

- **Service Autonomy (297)** – *Services exercise a high level of control over their underlying runtime execution environment.*

  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. Service Autonomy (297) supports the extent to which other design principles can be effectively realized in real-world production environments.

- **Service Statelessness (298)** – *Services minimize resource consumption by deferring the management of state information when necessary.*

  The management of excessive state information can compromise the availability of a service as well as the predictability of its behavior. Services are therefore ideally designed to remain stateful only when required. Like Service Autonomy (297), this is another principle that focuses less on the contract and more on the design of the underlying logic.

- **Service Discoverability (300)** – *Services are supplemented with communicative metadata by which they can be effectively discovered and interpreted.*

  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 service contracts and capabilities into account, regardless of whether a discovery mechanism such as a service registry is an immediate part of the environment.

- **Service Composability (302)** – *Services are effective composition participants, regardless of the size and complexity of the composition.*

As the sophistication of service-oriented solutions grows, so does the complexity of underlying service composition configurations. The ability to effectively compose services is a critical requirement for achieving some of the fundamental goals of service-oriented computing. Complex service compositions place demands on service design. Services are expected to be capable of participating as effective composition members, regardless of whether they need to be immediately enlisted in a composition.

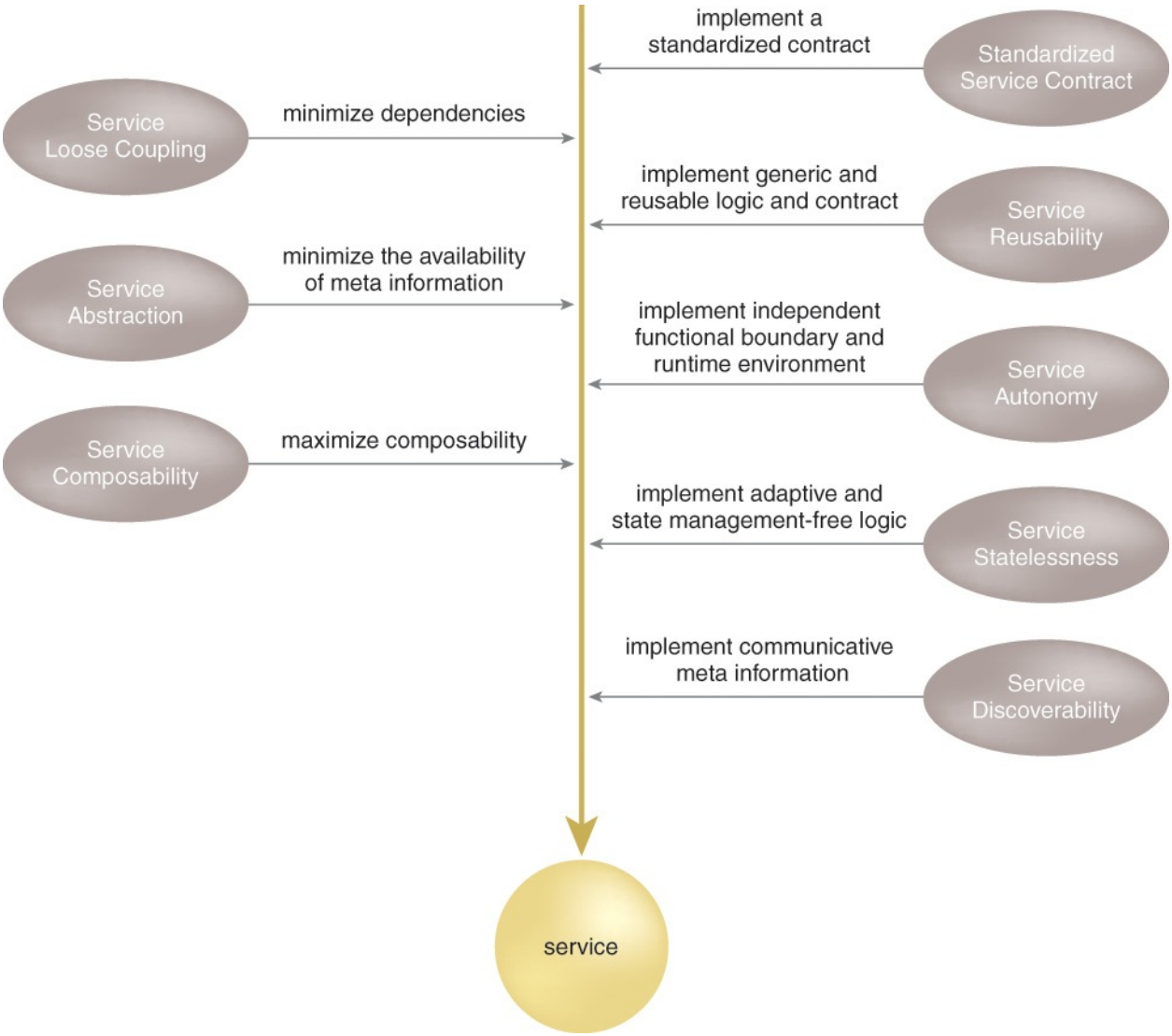


Figure 3.9 How service-orientation design principles collectively shape service design.

SOA Patterns

Service-orientation principles are closely related to SOA patterns.

Note how each pattern profile table in Appendix C contains a field dedicated to showing related design principles.

3.2 Problems Solved by Service-Orientation

To best appreciate why service-orientation 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.

Silo-based Application Architecture

In the world of business, delivering solutions capable of automating the execution of business tasks makes a great deal of sense. 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 3.10).

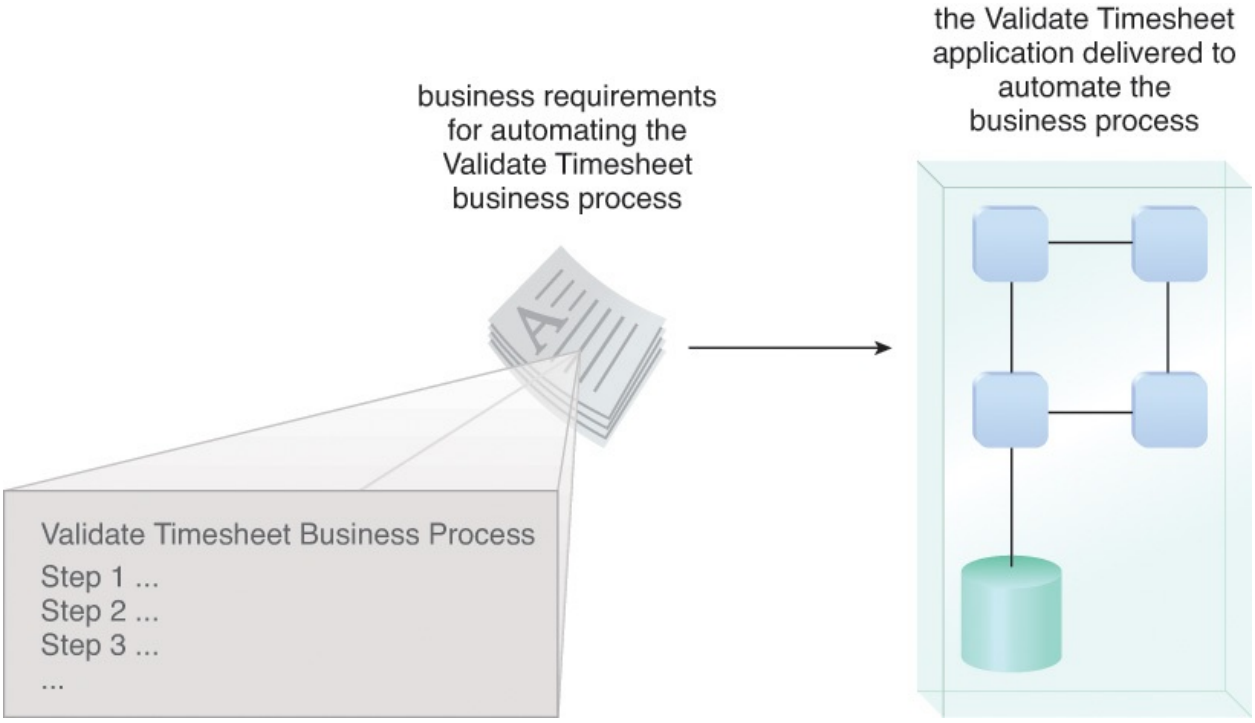
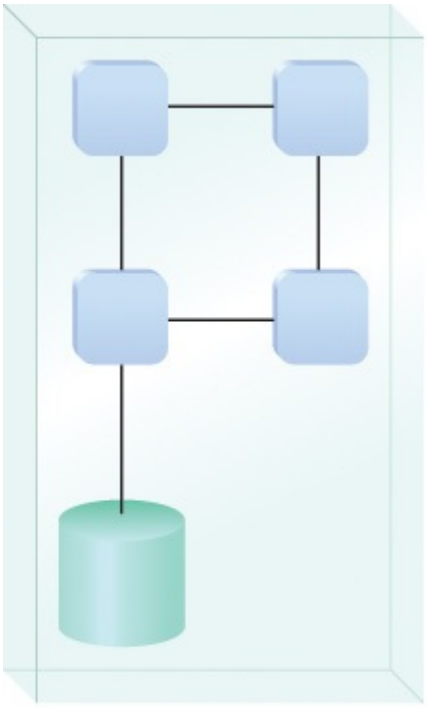


Figure 3.10 A ratio of one application for each new set of automation requirements has been common.

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 3.11).



Validate Timesheet Application

Development cost = x

Yearly operational cost = y

Estimated yearly savings due to increased productivity = (x/2) - y

Figure 3.11 A sample formula for calculating ROI is based on a predetermined investment with a predictable return.

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 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 straightforward. 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 much 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 3.12). The effort and expense required to construct this logic is therefore also redundant.

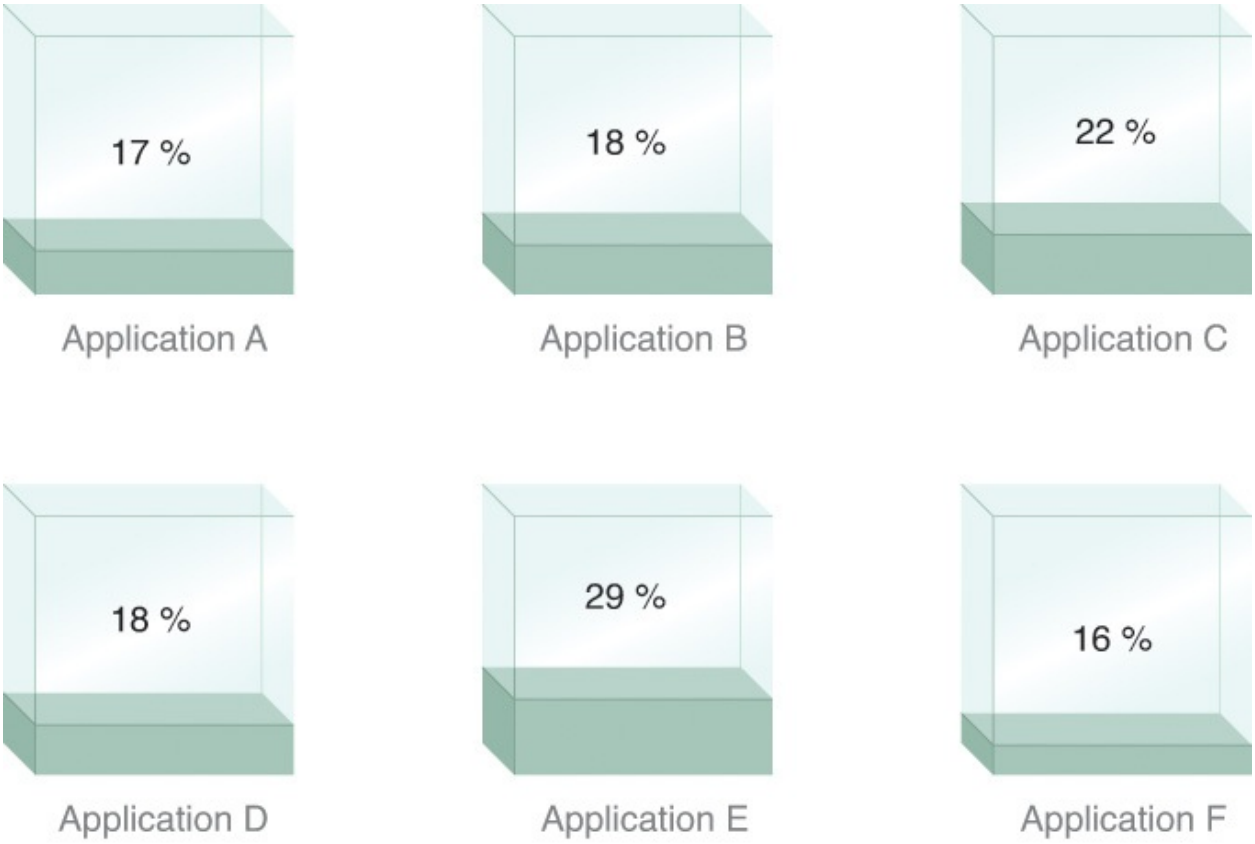


Figure 3.12 Different applications developed independently can result in significant amounts of redundant functionality. The applications displayed were delivered with various levels of solution logic that, in some form, already existed.

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 3.13).

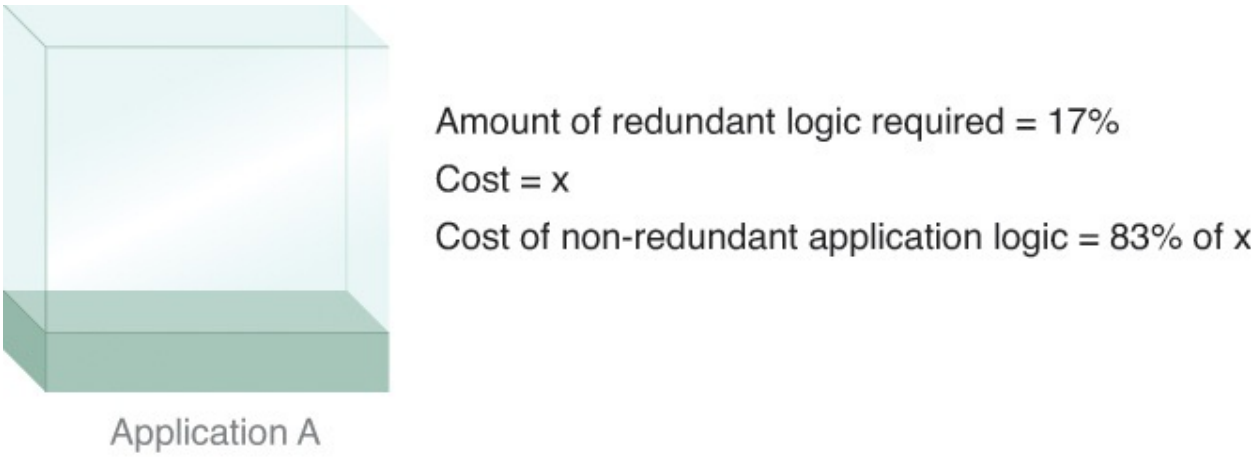


Figure 3.13 Application A was delivered for a specific set of business requirements. Because a subset of these business requirements had already been fulfilled elsewhere, Application A's delivery scope is larger than it has to be.

It Bloats an Enterprise

Each new or extended application adds to the bulk of an IT environment's system inventory (Figure 3.14). 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.

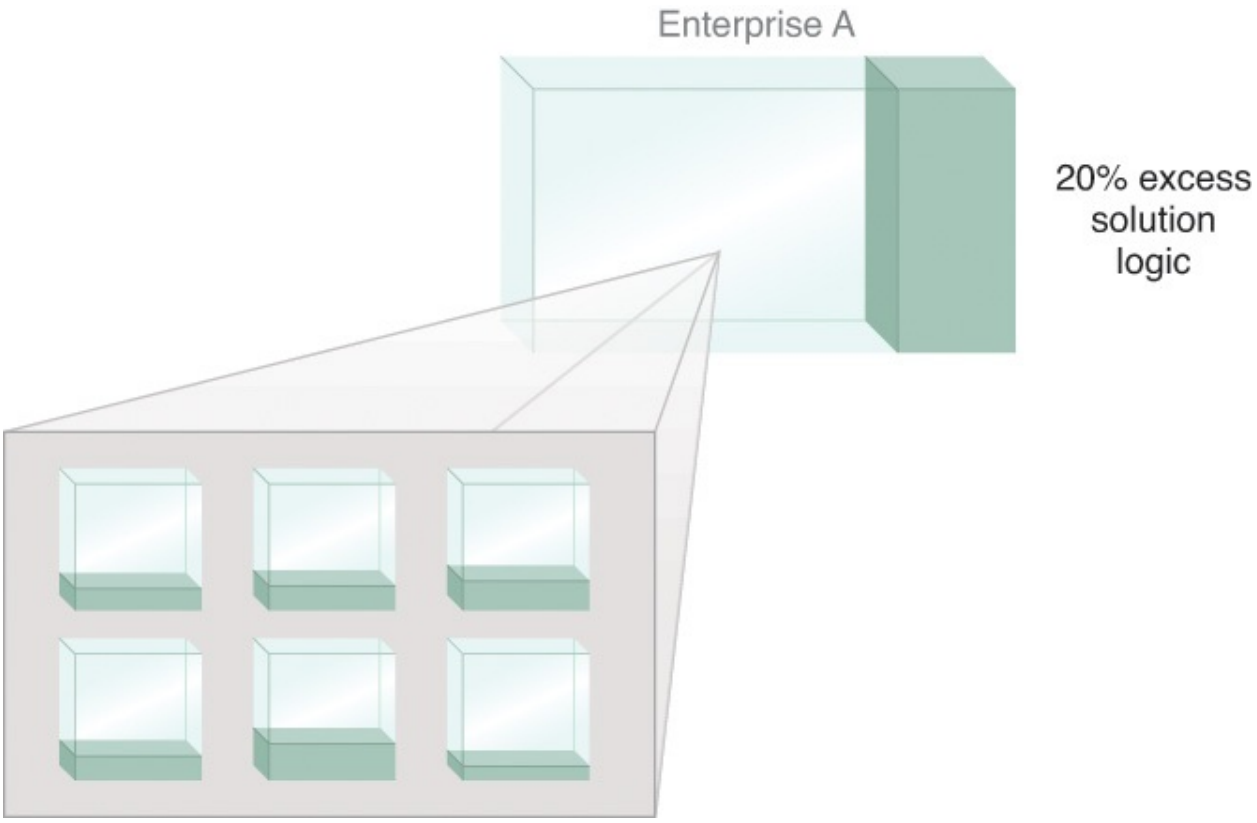


Figure 3.14 This simple diagram portrays an enterprise environment containing applications with redundant functionality. The net effect is a larger enterprise.

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 3.15), making it challenging to plan the evolution of an enterprise and scale its infrastructure in response to that evolution.

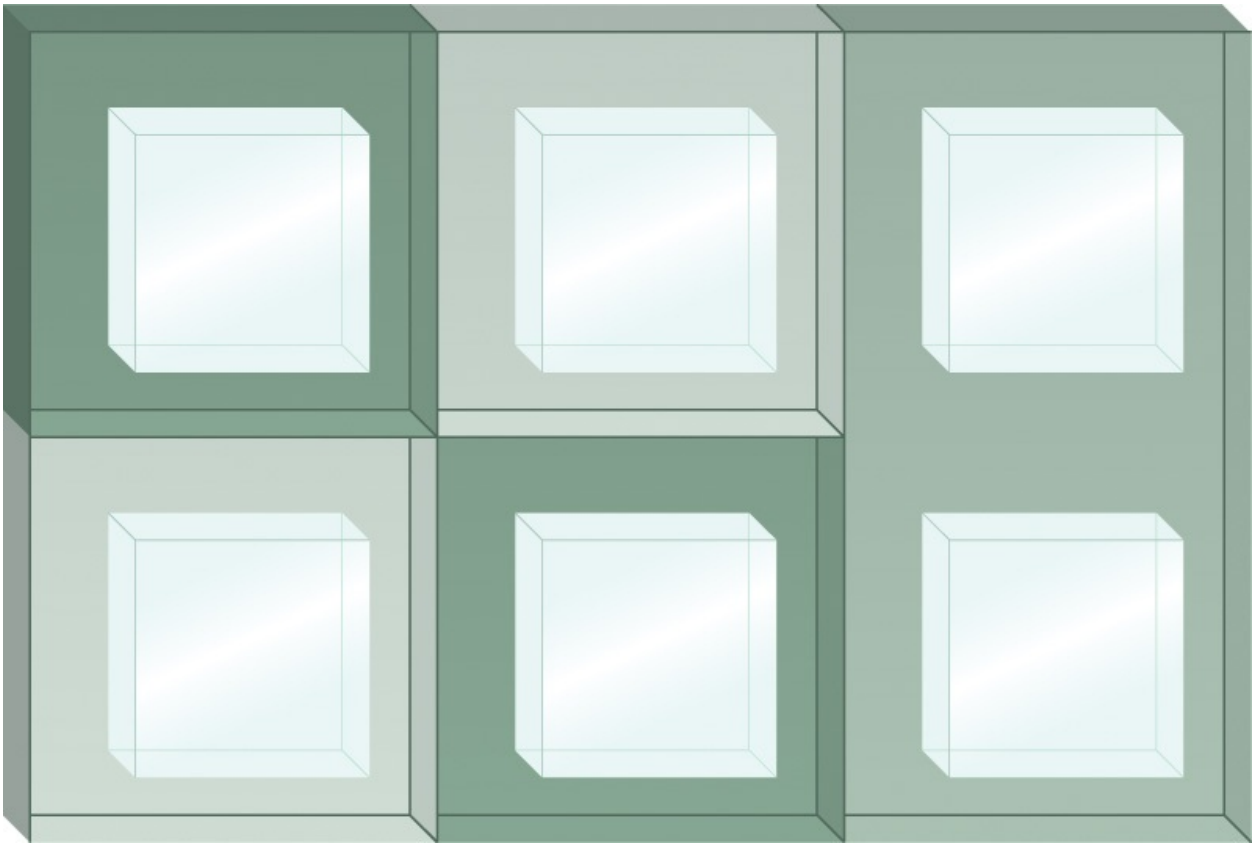


Figure 3.15 Different application environments within the same enterprise can introduce incompatible runtime platforms as indicated by the shaded zones.

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 3.16) or requiring the introduction of large middleware layers.

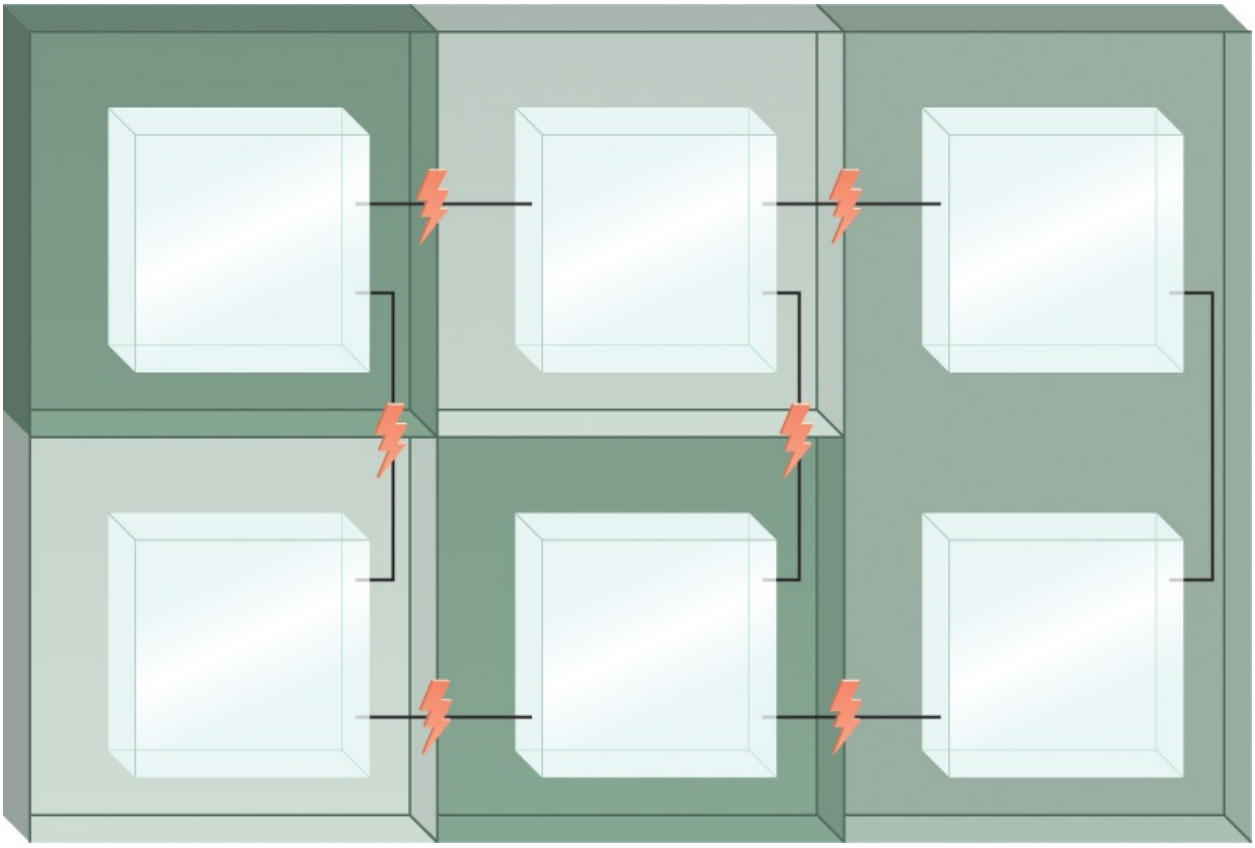


Figure 3.16 A vendor-diverse enterprise can introduce a variety of integration challenges, as expressed by the little lightning bolts that highlight points of concern when trying to bridge proprietary environments.

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 we 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 Reusable Solution Logic

Within a service-oriented solution, units of logic (services) encapsulate functionality not specific to any one application or business process (Figure 3.17). These services are therefore classified as reusable (and agnostic) IT assets.

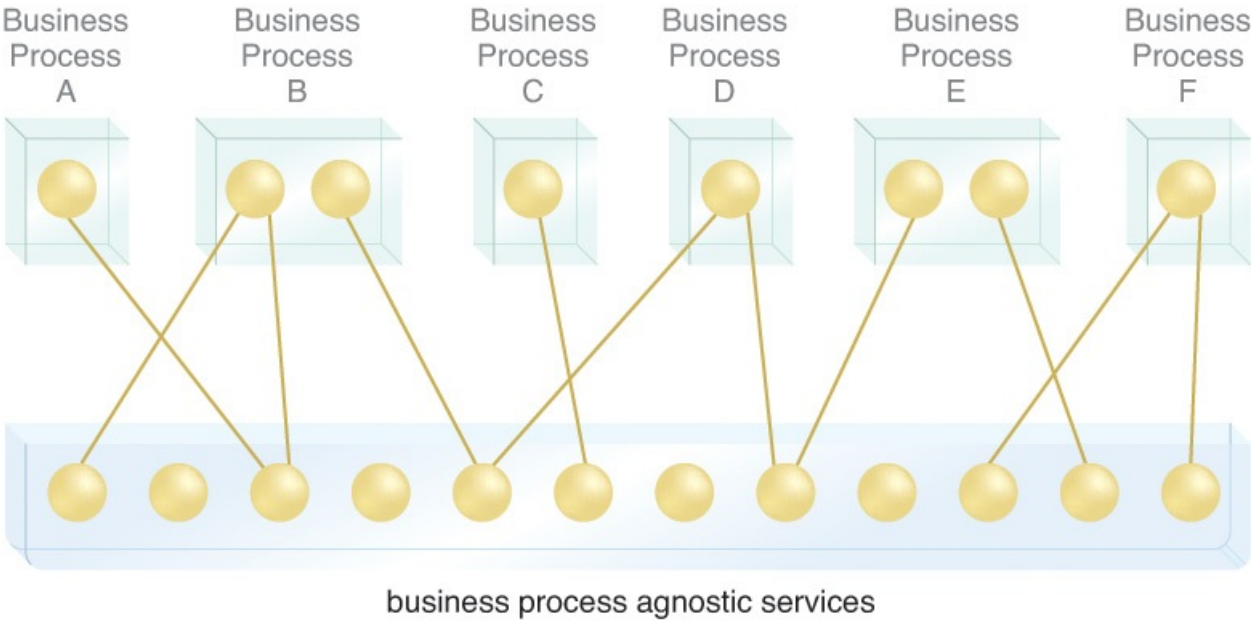


Figure 3.17 Business processes are automated by a series of business process–specific services (top layer) that share a pool of business process–agnostic services (bottom layer). These layers correspond to service models described in Chapter 5.

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 (or "non-agnostic") logic (Figure 3.18). This blurs the lines between standalone application environments by reducing the overall quantity of standalone

applications. (See the *Service-Orientation and the Concept of "Application"* section later in this chapter.)

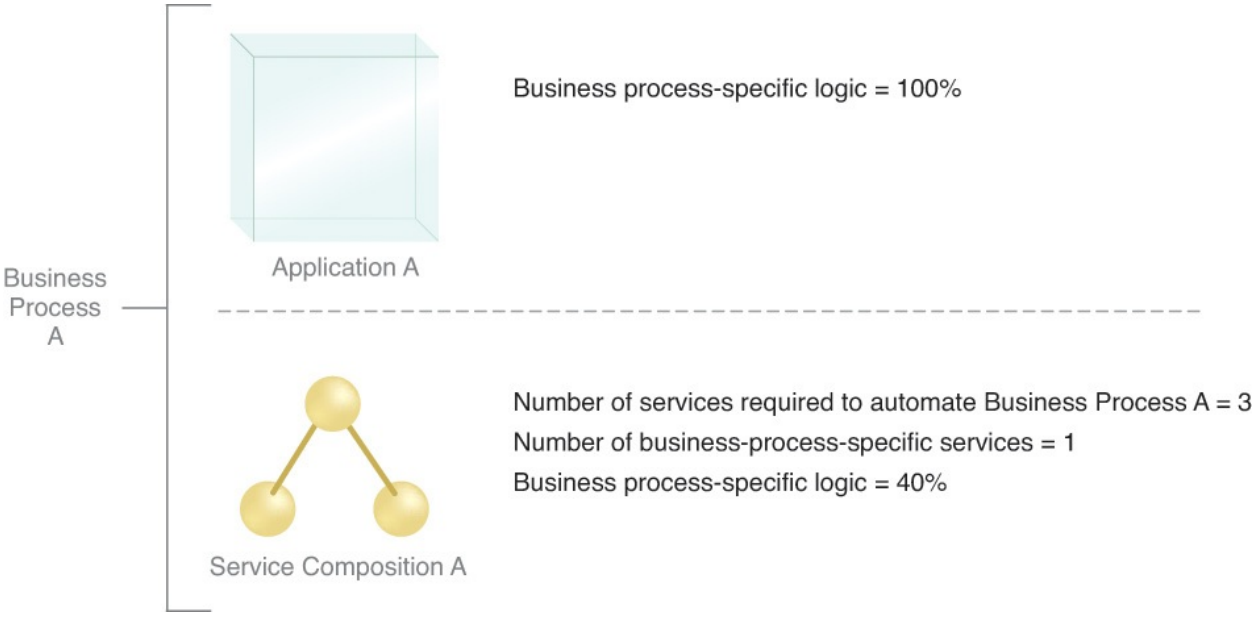


Figure 3.18 Business Process A can be automated by either Application A or Service Composition A. The delivery of Application A can result in a body of solution logic that is all specific to and tailored for the business process. Service Composition A would be designed to automate the process with a combination of reusable 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 3.19.

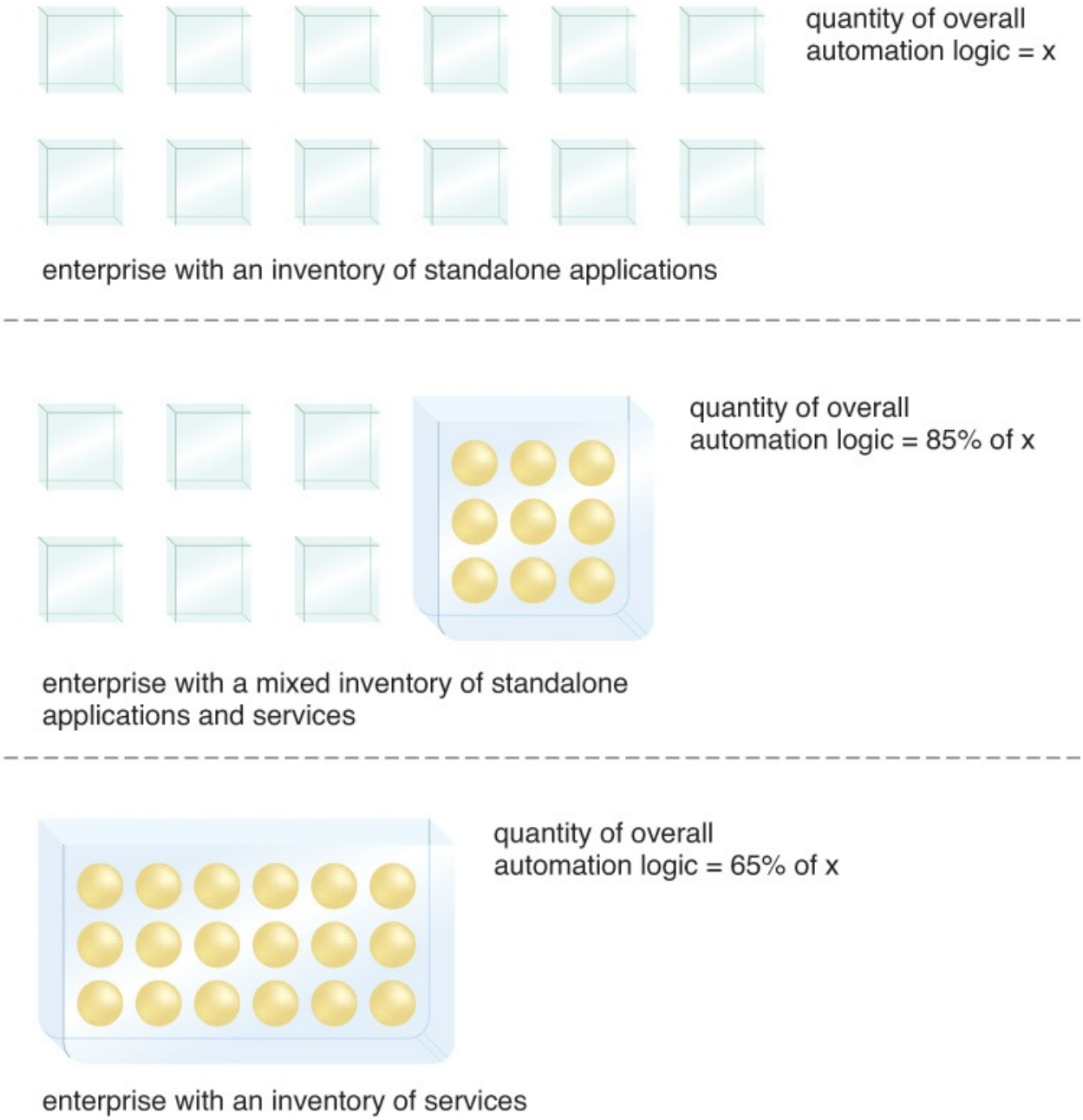


Figure 3.19 The quantity of solution logic shrinks as an enterprise transitions toward a standardized service inventory comprised of "normalized" services. (Service normalization is explained further at the end of Chapter 5.)

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 3.20. (See the

Service-Orientation and the Concept of "Integration" section later in this chapter.)

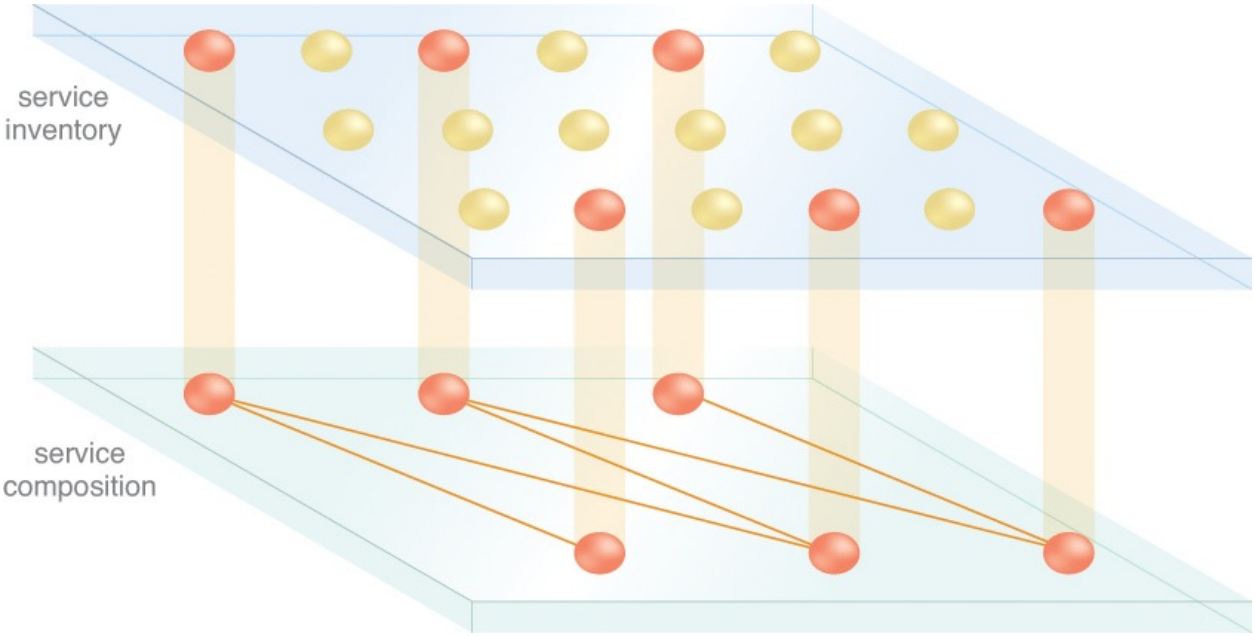


Figure 3.20 Services from different parts of a service inventory can be combined into new compositions. If these services are designed to be intrinsically interoperable, the effort to assemble them into new composition configurations is significantly reduced.

Note

See Chapter 4 in *SOA Principles of Service Design* for coverage of common challenges introduced by service-orientation.

3.3 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 or 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 3.21). What was an application is now just another composition of services, some of which likely participate in other compositions (Figure 3.22).

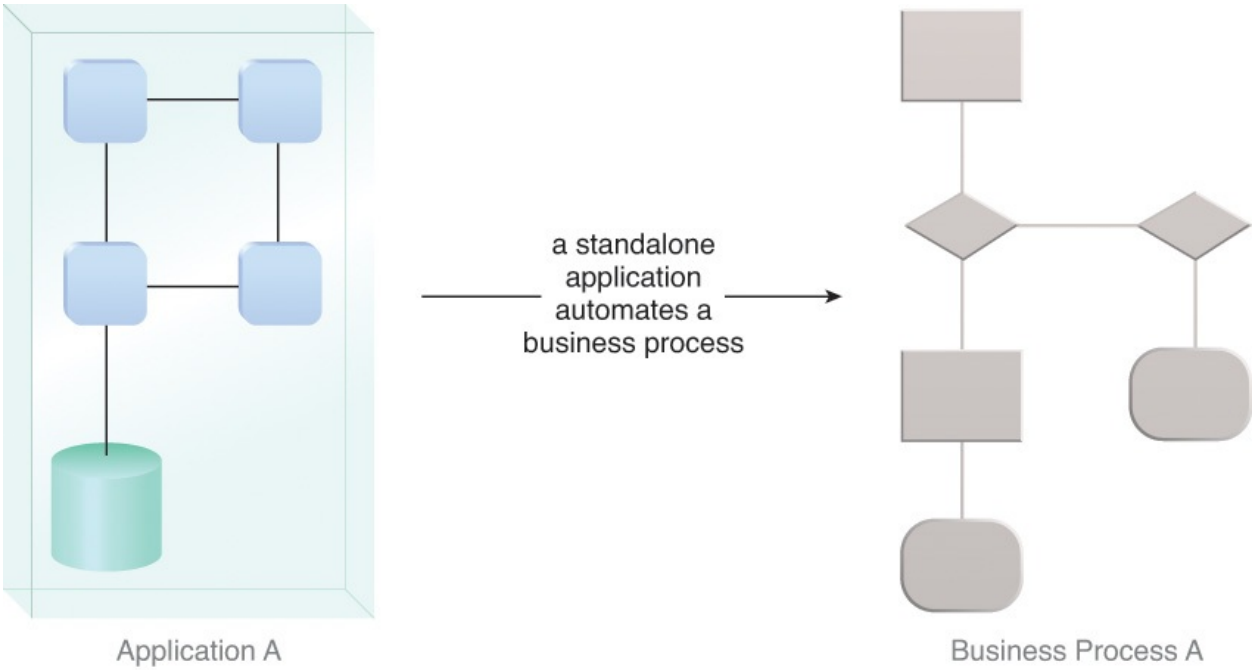


Figure 3.21 The traditional application, delivered to automate specific business process logic.

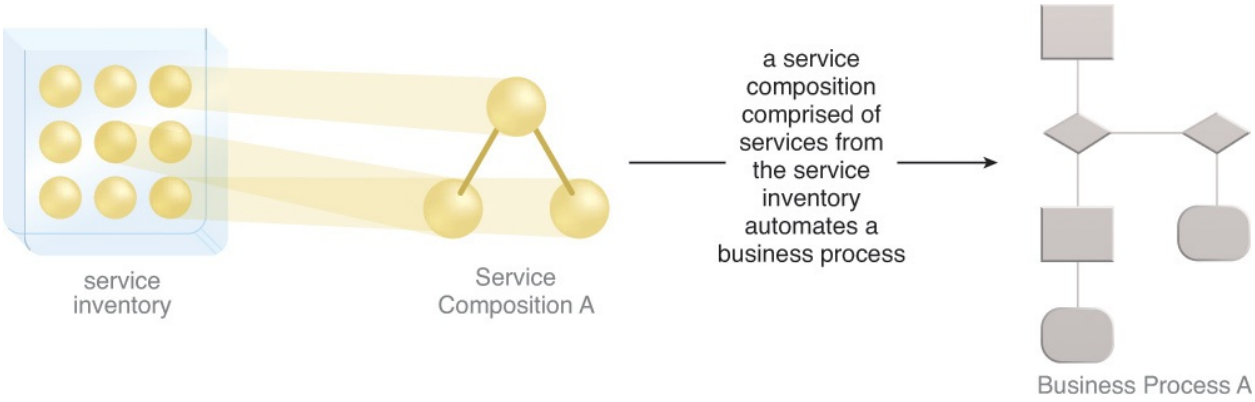


Figure 3.22 The service composition, intended to fulfill the role of the traditional application by leveraging agnostic and non-agnostic services from a service inventory. This essentially establishes a "composite application."

The application therefore 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 our services (based on the service models established in Chapter 5) are actually not business process agnostic. One service, for example, intentionally represents logic that is dedicated to the automation of just one business task, and therefore not necessarily reusable.

So, single-purpose 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 will 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 3.23). 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.

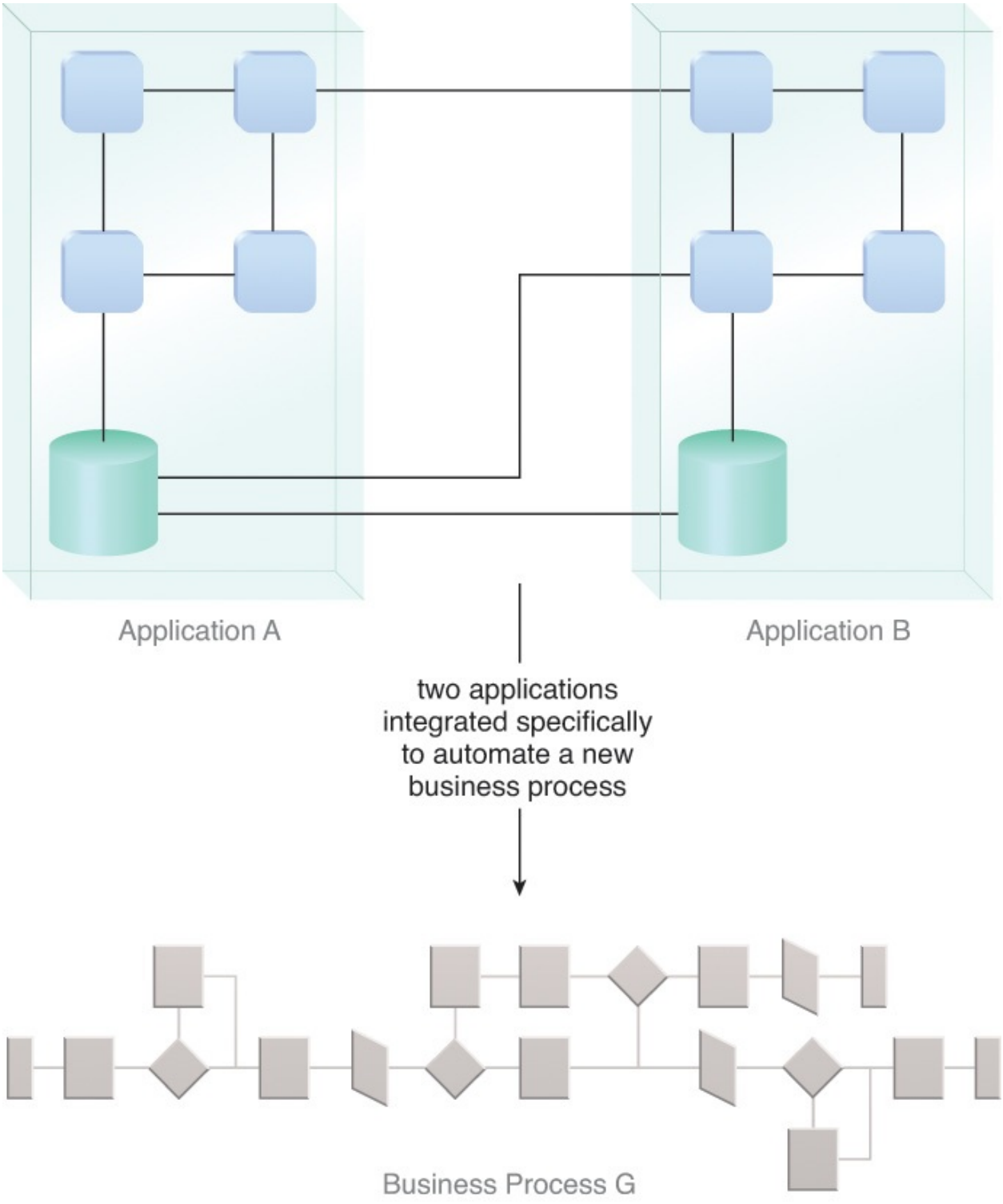


Figure 3.23 The traditional integration architecture, comprised of two or more applications connected in different ways to fulfill a new set of automation requirements (as dictated by the new Business Process G).

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 3.24). 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 but also between legacy systems and these services.

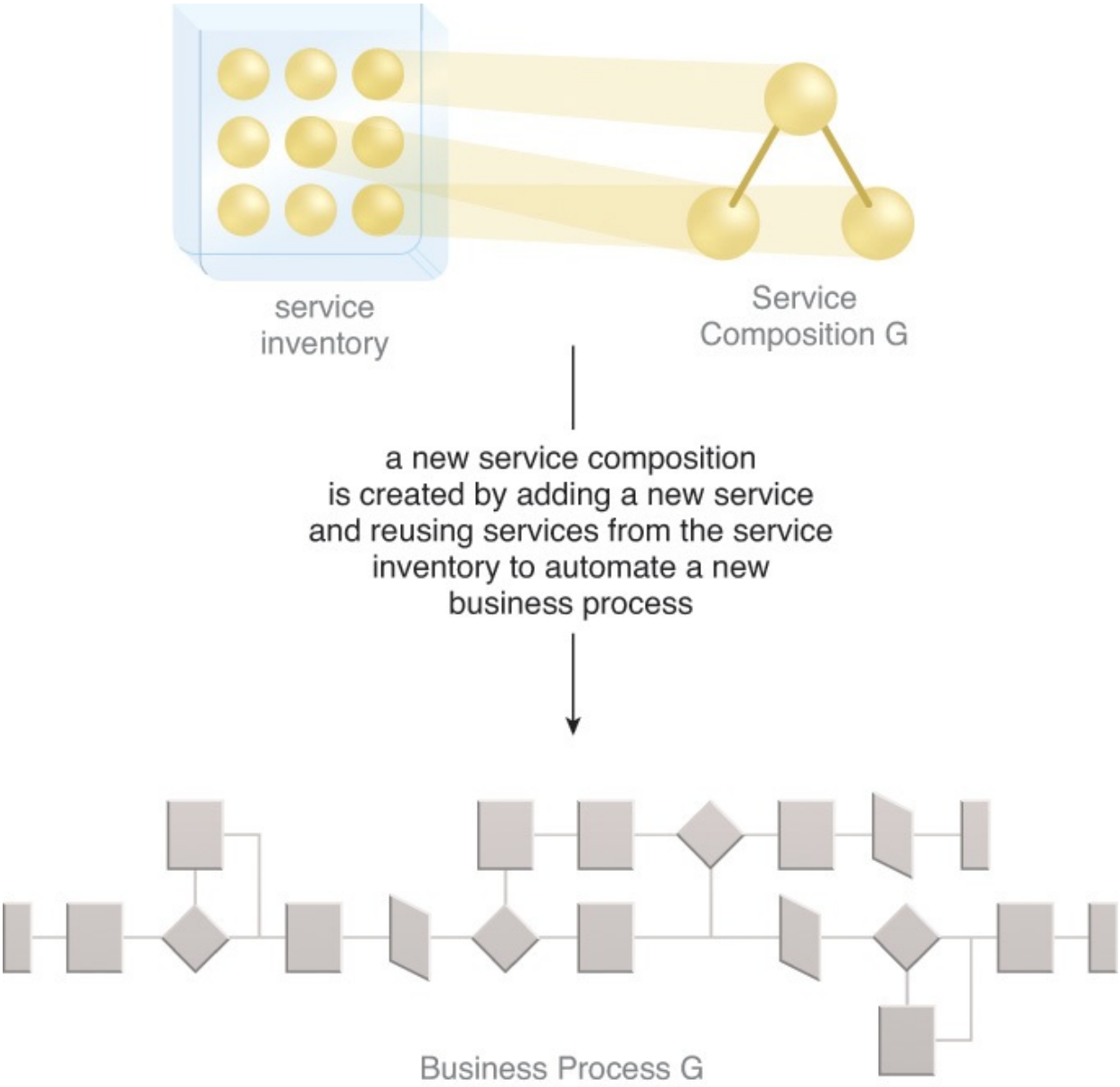


Figure 3.24 A new combination of services is composed together to fulfill the role of traditional integrated applications.

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 3.25). 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.

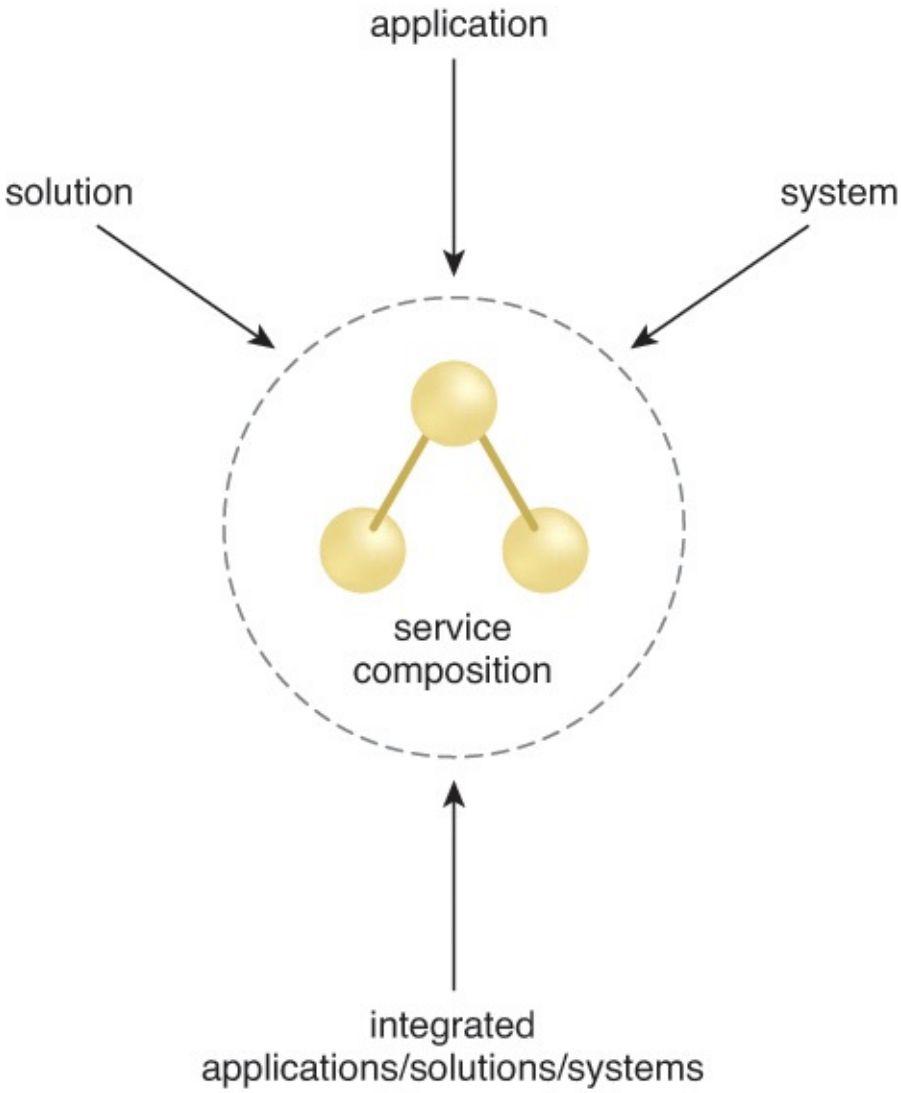


Figure 3.25 A service-oriented solution, application, or system is the equivalent of a service composition.

3.4 Goals and Benefits of Service-Oriented Computing

A set of strategic goals and benefits (Figure 3.26) collectively represents the target state we look to achieve when we consistently apply service-orientation to the design of software programs. It is highly beneficial to understand the significance of these goals and benefits because they provide us with constant, overarching context and justification for maintaining our commitment to carrying out service-orientation over the long term.

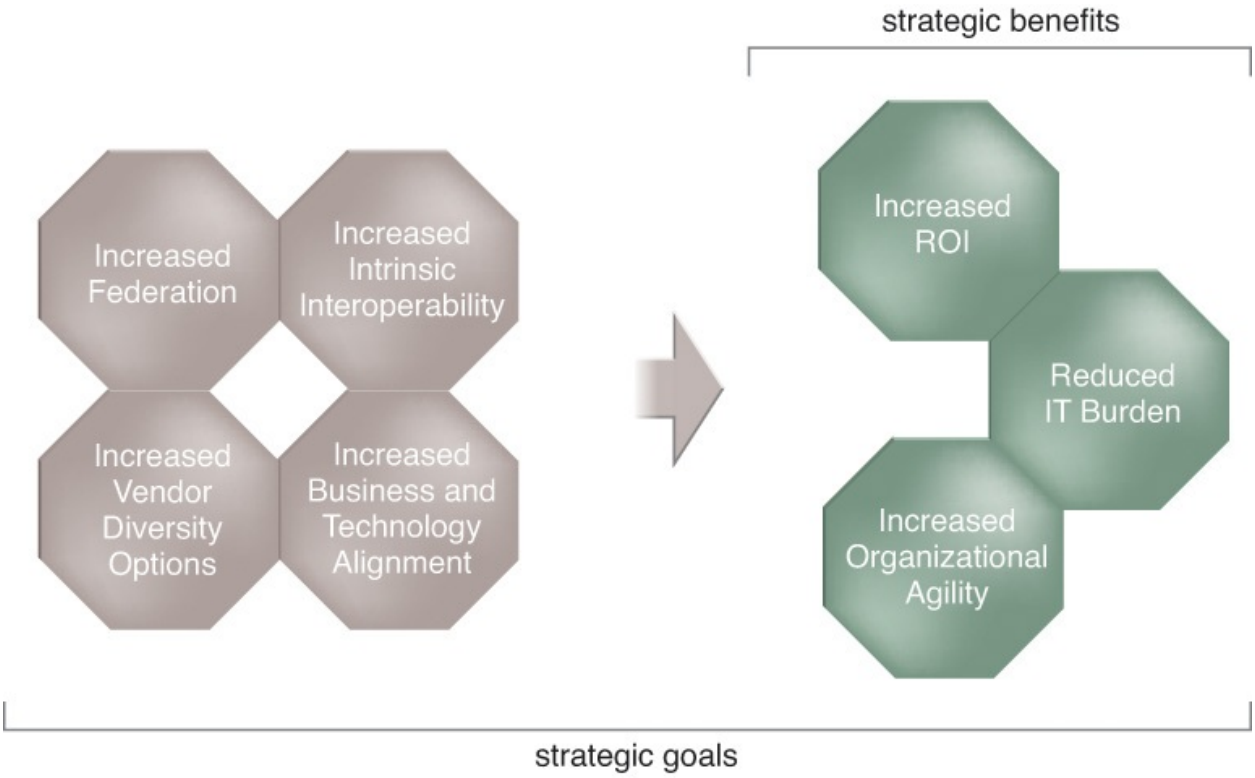


Figure 3.26 The seven identified goals are interrelated and can be further categorized into two groups: strategic goals and resulting benefits. Increased organization agility, increased ROI, and reduced IT burden are concrete benefits resulting from the attainment of the remaining four goals.

The upcoming sections describe each of these strategic goals and benefits.

Increased Intrinsic Interoperability

Interoperability refers to the sharing of data. The more interoperable software programs are, the easier it is for them to exchange information. Software programs that are not interoperable need to be integrated. Therefore, integration can be seen as a process that enables interoperability. A goal of service-orientation is to establish native interoperability within services to reduce the need for integration (Figure 3.27). As previously explained in the *Effects of Service-Orientation on the Enterprise* section, integration as a concept begins to fade within service-oriented environments.

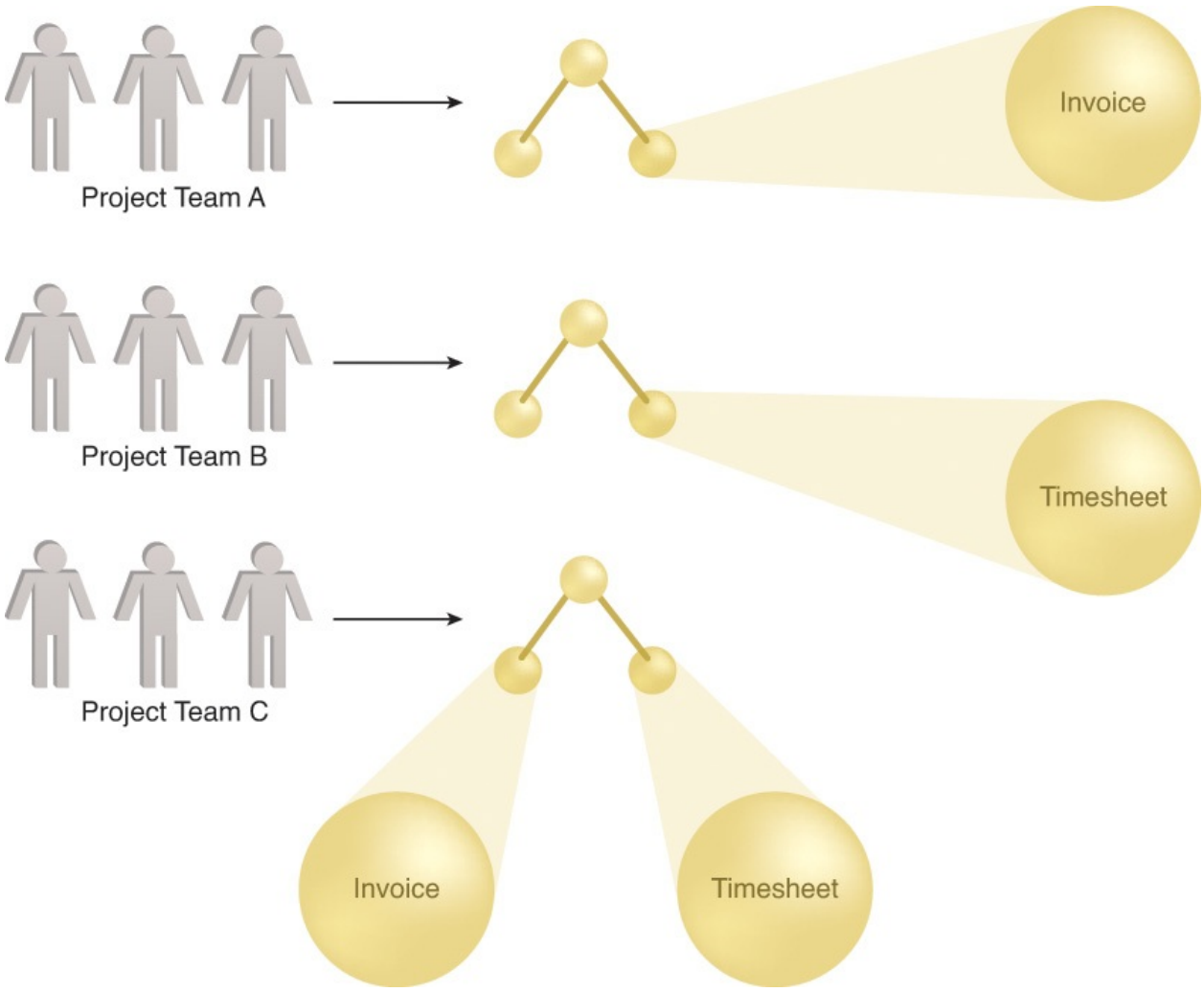


Figure 3.27 Services are designed to be intrinsically interoperable regardless of when and for which purpose they are delivered. In this example, the intrinsic interoperability of the Invoice and Timesheet services delivered by Project Teams A and B allow them to be combined into a new service composition by Project Team C.

Interoperability is specifically fostered through the consistent application of design principles and design standards. This establishes an environment wherein services produced by different projects at different times can be repeatedly assembled together into a variety of composition configurations to help automate a range of business tasks.

Intrinsic interoperability represents a fundamental goal of service-orientation that establishes a foundation for the realization of other strategic goals and benefits. Contract standardization, scalability, behavioral predictability, and reliability are just some of the design characteristics required to facilitate interoperability, all of which are addressed by the service-orientation principles

documented in this book.

Each of the eight service-orientation principles supports or contributes to interoperability in some manner. The following are just a few examples:

- *Standardized Service Contract (291)* – Service contracts are standardized to guarantee a baseline measure of interoperability associated with the harmonization of data models.
- *Service Loose Coupling (293)* – 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.
- *Service Abstraction (294)* – 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.
- *Service Reusability (295)* – Designing services for reuse implies a high-level of required interoperability between the service and numerous potential service consumers.
- *Service Autonomy (297)* – By raising a service's individual autonomy its behavior becomes more consistently predictable, increasing its reuse potential and thereby its attainable level of interoperability.
- *Service Statelessness (298)* – Through an emphasis on stateless design, the availability and scalability of services increase, allowing them to interoperate more frequently and reliably.
- *Service Discoverability (300)* – Being discoverable simply allows services to be more easily located by those who want to potentially interoperate with them.
- *Service Composability (302)* – 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.

Increased Federation

A federated IT environment is one where resources and applications are united while maintaining their individual autonomy and self-governance. Service-orientation aims to increase a federated perspective of an enterprise to whatever extent it is applied. It accomplishes this through the widespread deployment of standardized and composable services, each of which encapsulates a segment of the enterprise and expresses it in a consistent manner.

In support of increasing federation, standardization becomes part of the extra upfront attention each service receives at design time. Ultimately this leads to an environment where enterprise-wide solution logic becomes naturally harmonized, regardless of the nature of its underlying implementation (Figure 3.28).

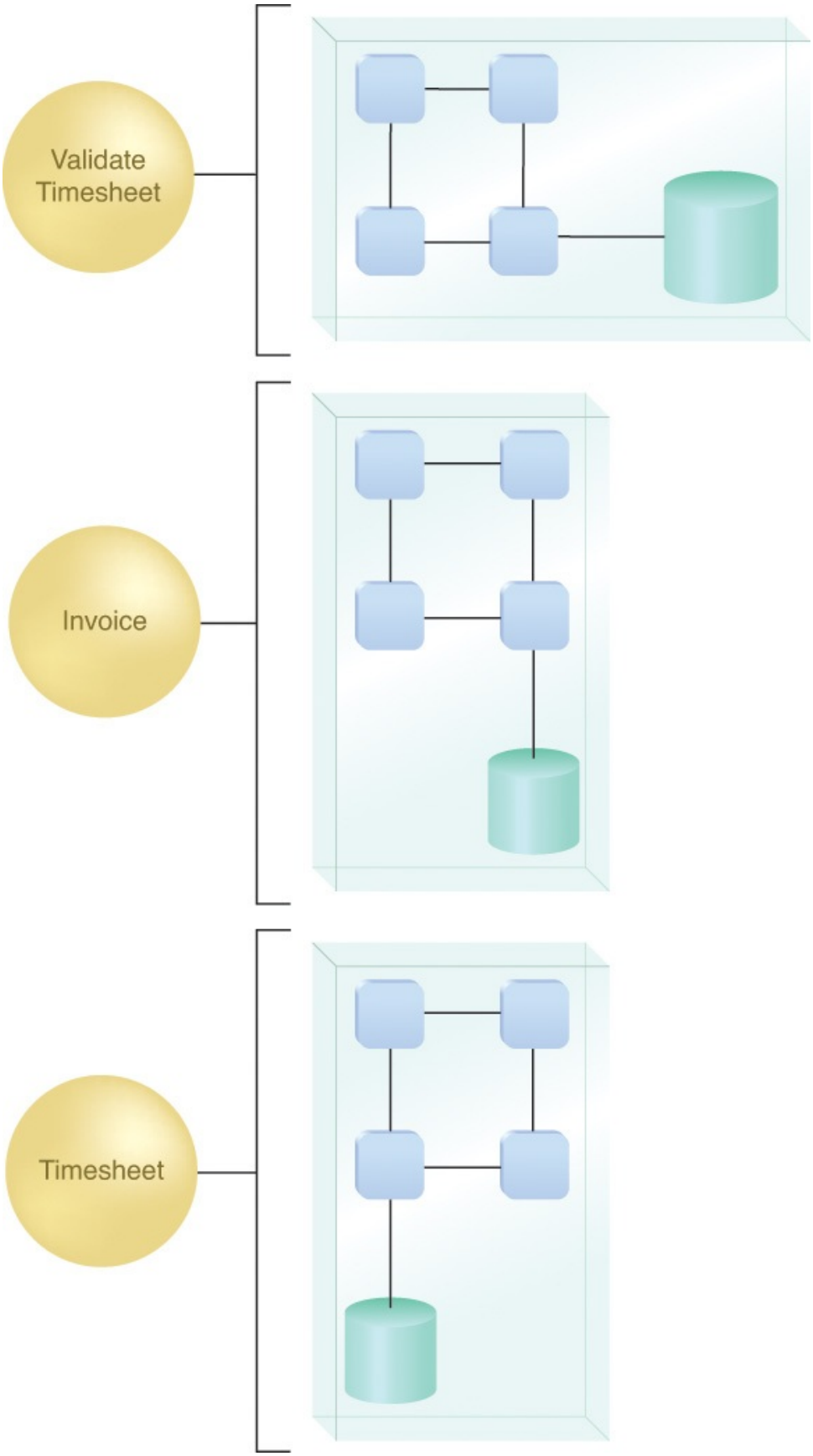

Validate Timesheet
Invoice
Timesheet

Figure 3.28 Three service contracts establishing a federated set of endpoints, each of which encapsulates a different implementation.

Increased Vendor Diversification Options

Vendor diversification refers to the ability an organization has to pick and choose "best-of-breed" vendor products and technology innovations and use them together within one enterprise. Having a vendor-diverse environment is not necessarily beneficial for an organization; however, having the *option* to diversify when required is beneficial. To have and retain this option requires that its technology architecture not be tied or locked into any one specific vendor platform.

This represents an important state for an enterprise in that it provides the constant freedom for an organization to change, extend, and even replace solution implementations and technology resources without disrupting the overall, federated service architecture. This measure of governance autonomy is attractive because it prolongs the lifespan and increases the financial return of automation solutions.

By designing a service-oriented solution in alignment with but neutral to major vendor SOA platforms and by positioning service contracts as standardized endpoints throughout a federated enterprise, proprietary service implementation details can be abstracted to establish a consistent interservice communications framework. This provides organizations with constant options by allowing them to diversify their enterprise as needed (Figure 3.29).

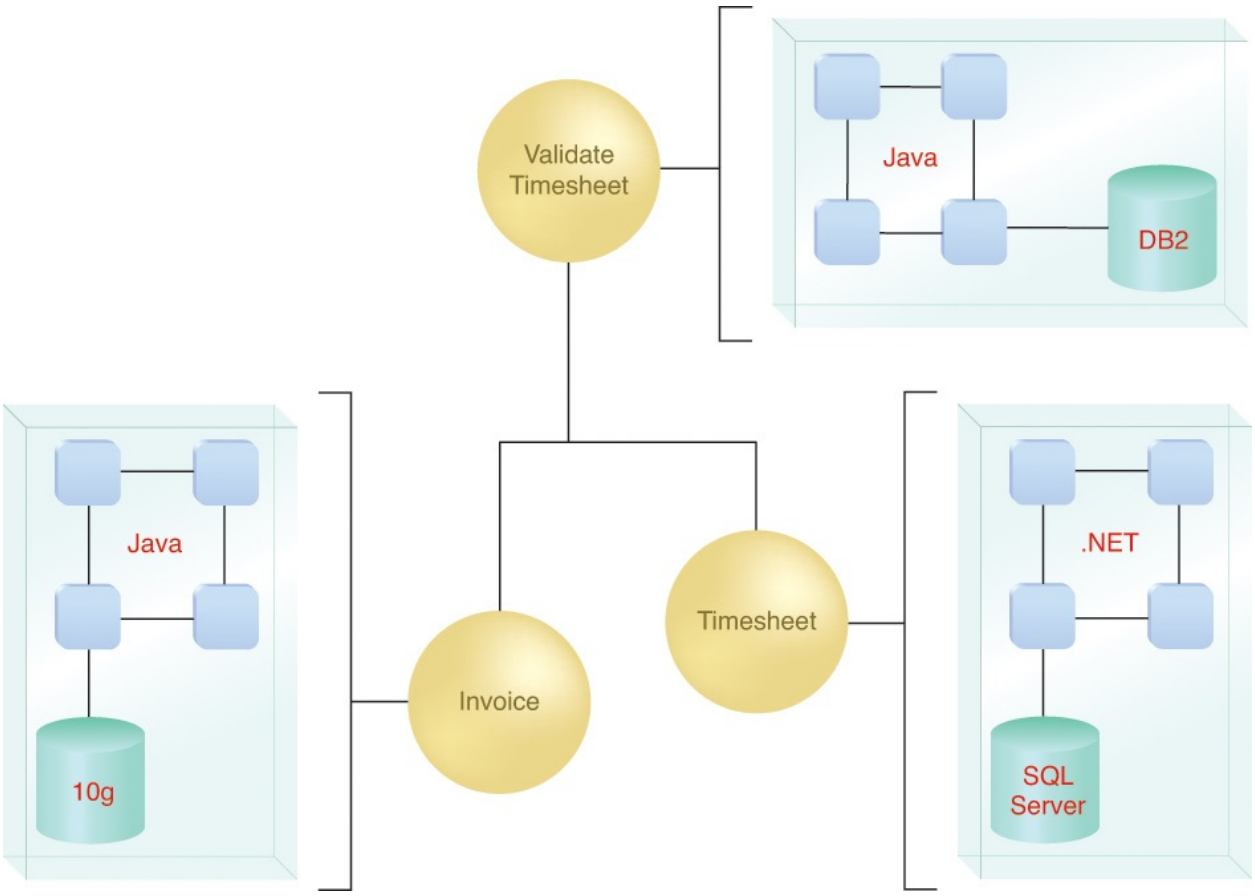


Figure 3.29 A service composition consisting of three services, each of which encapsulates a different vendor automation environment. If service-orientation is adequately applied to the services, underlying disparity will not inhibit their ability to be combined into effective compositions.

Vendor diversification is further supported by taking advantage of the standards-based, vendor-neutral Web services framework. Because they impose no proprietary communication requirements, services further decrease dependency on vendor platforms. As with any other implementation medium, though, services need to be shaped and standardized through service-orientation to become a federated part of a greater service inventory.

Increased Business and Technology Domain Alignment

The extent to which IT business requirements are fulfilled is often associated with the accuracy with which business logic is expressed and automated by solution logic. Although initial application implementations have traditionally been designed to meet initial requirements, there has historically been a challenge in keeping applications in alignment with business needs as the nature and direction of the business changes.

Service-orientation promotes abstraction on many levels. One of the most effective means by which functional abstraction is applied is the establishment of service layers that accurately encapsulate and represent business models. By doing so, common, pre-existing representations of business logic (business entities, business processes) can exist in implemented form as physical services.

This is accomplished by incorporating a structured analysis and modeling process that requires the hands-on involvement of business subject matter experts in the actual definition of the services (as explained in the *Service-Oriented Analysis (Service Modeling)* section in Chapter 4). The resulting service designs are capable of aligning automation technology with business intelligence on an unprecedented level (Figure 3.30).

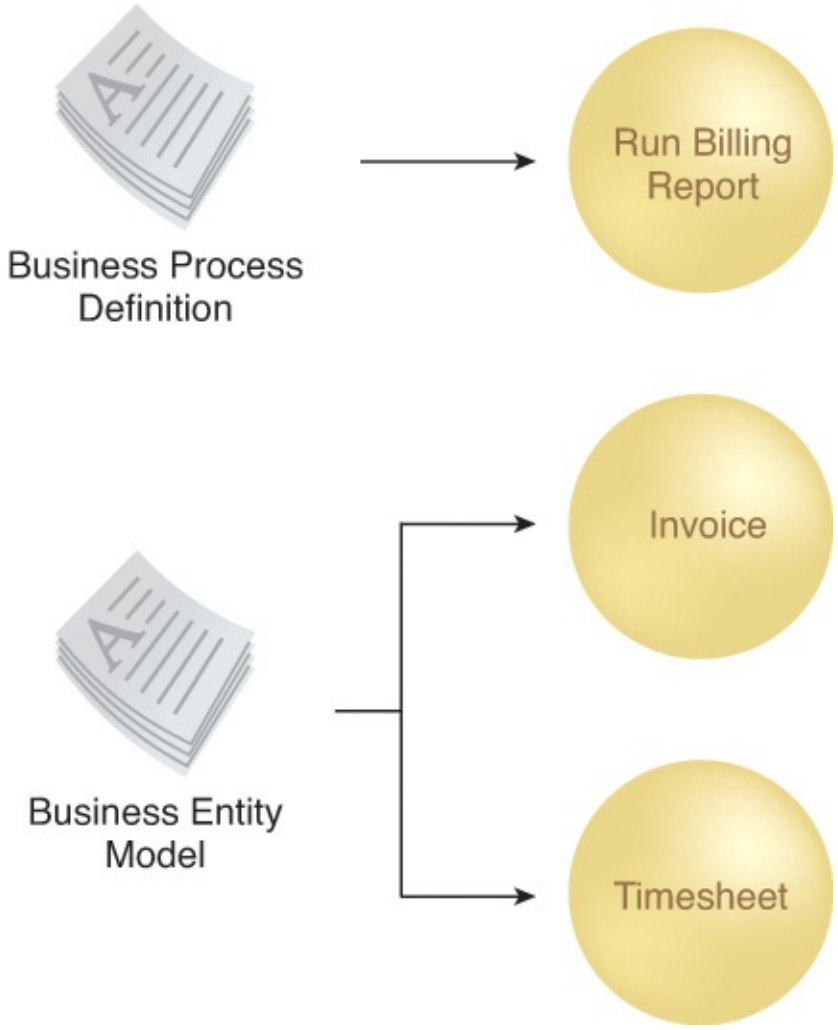


Figure 3.30 Services with business-centric functional contexts are carefully modeled to express and encapsulate corresponding business models and logic.

Furthermore, the fact that services are designed to be intrinsically interoperable

directly facilitates business change. As business processes are augmented in response to various factors (business climates, new policies, new priorities, etc.) services can be reconfigured into new compositions that reflect the changed business logic. This allows a service-oriented technology architecture to evolve in tandem with the business itself.

Increased ROI

Measuring the return on investment (ROI) of automated solutions is a critical factor in determining just how cost effective a given application or system actually is. The greater the return, the more an organization benefits from the solution. However, the lower the return, the more the cost of automated solutions eats away at an organization's budgets and profits.

Because the nature of required application logic has increased in complexity and due to ever-growing, non-federated integration architectures that are difficult to maintain and evolve, the average IT department represents a significant amount of an organization's operational budget. For many organizations, the financial overhead required by IT is a primary concern because it often continues to rise without demonstrating any corresponding increase in business value.

Service-orientation advocates the creation of agnostic solution logic—logic that is agnostic to any one purpose and therefore useful for multiple purposes. This multipurpose or reusable logic fully leverages the intrinsically interoperable nature of services. Agnostic services have increased reuse potential that can be realized by allowing them to be repeatedly assembled into different compositions. Any one agnostic service can therefore find itself being repurposed numerous times to automate different business processes as part of different service-oriented solutions.

With this benefit in mind, additional upfront expense and effort is invested into every piece of solution logic to position it as an IT asset for the purpose of repeatable, long-term financial returns. As shown in Figure 3.31, the emphasis on increasing ROI typically goes beyond the returns traditionally sought as part of past reuse initiatives. This has much to do with the fact that service-orientation aims to establish reuse as a common, secondary characteristic within most services.

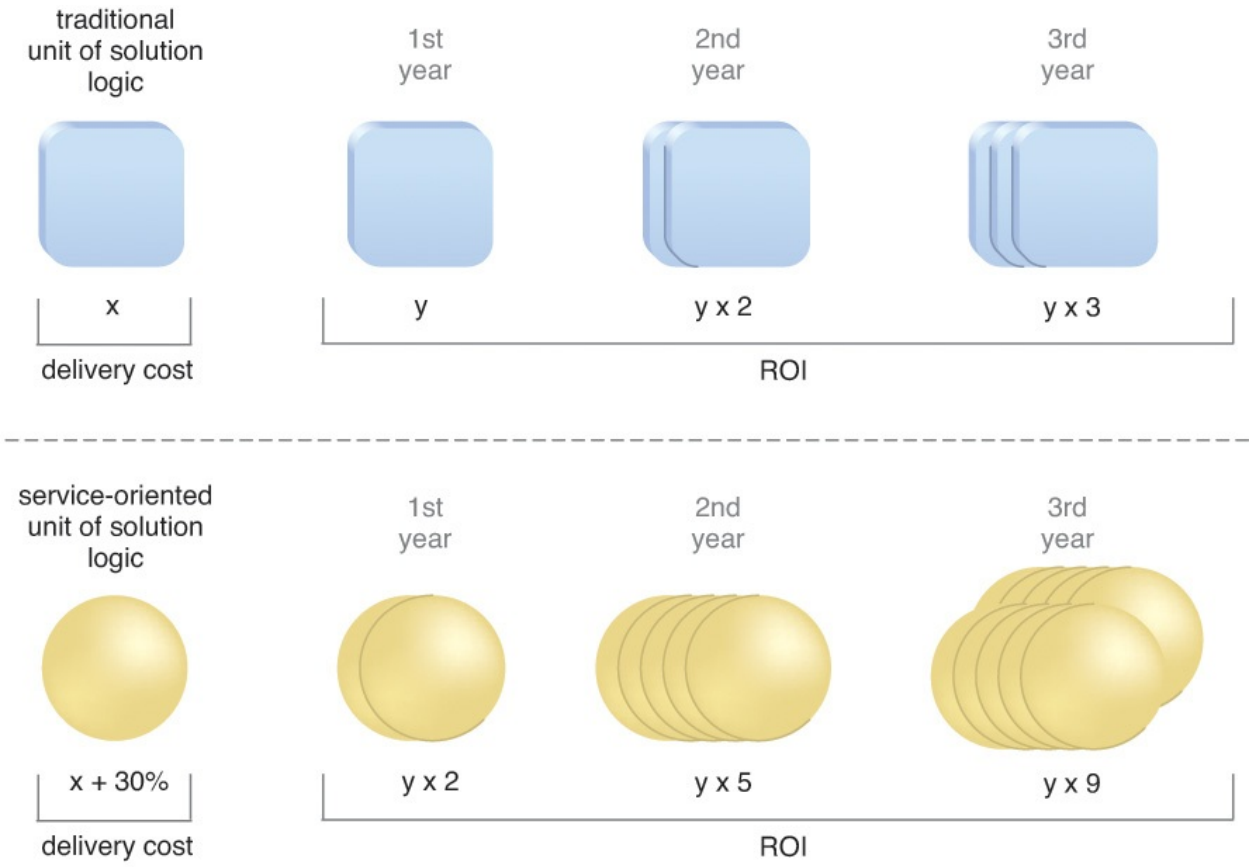


Figure 3.31 An example of the types of formulas being used to calculate ROI for SOA projects. More is invested in the initial delivery with the goal of benefiting from increased subsequent reuse.

It is important to acknowledge that this goal is not simply tied to the benefits traditionally associated with software reuse. Proven commercial product design techniques are incorporated and blended with existing enterprise application delivery approaches to form the basis of a distinct set of service-oriented analysis and design processes (as covered in the chapters in Part II, *Service-Oriented Analysis and Design*).

Increased Organizational Agility

Agility, on an organizational level, refers to efficiency with which an organization can respond to change. Increasing organizational agility is very attractive to corporations, especially those in the private sector. Being able to more quickly adapt to industry changes and outmaneuver competitors has tremendous strategic significance.

An IT department can sometimes be perceived as a bottleneck, hampering desired responsiveness by requiring too much time or resources to fulfill new or

changing business requirements. This is one of the reasons agile development methods have gained popularity, as they provide a means of addressing immediate, tactical concerns more rapidly.

Service-orientation is very much geared toward establishing widespread organizational agility. When service-orientation is applied throughout an enterprise, it results in the creation of services that are highly standardized and reusable and therefore agnostic to parent business processes and specific application environments.

As a service inventory is comprised of more and more agnostic services, an increasing percentage of its overall solution logic belongs to no one application environment. Instead, because these services have been positioned as reusable IT assets, they can be repeatedly composed into different configurations. As a result, the time and effort required to automate new or changed business processes is correspondingly reduced because development projects can now be completed with significantly less custom development effort (Figure 3.32).

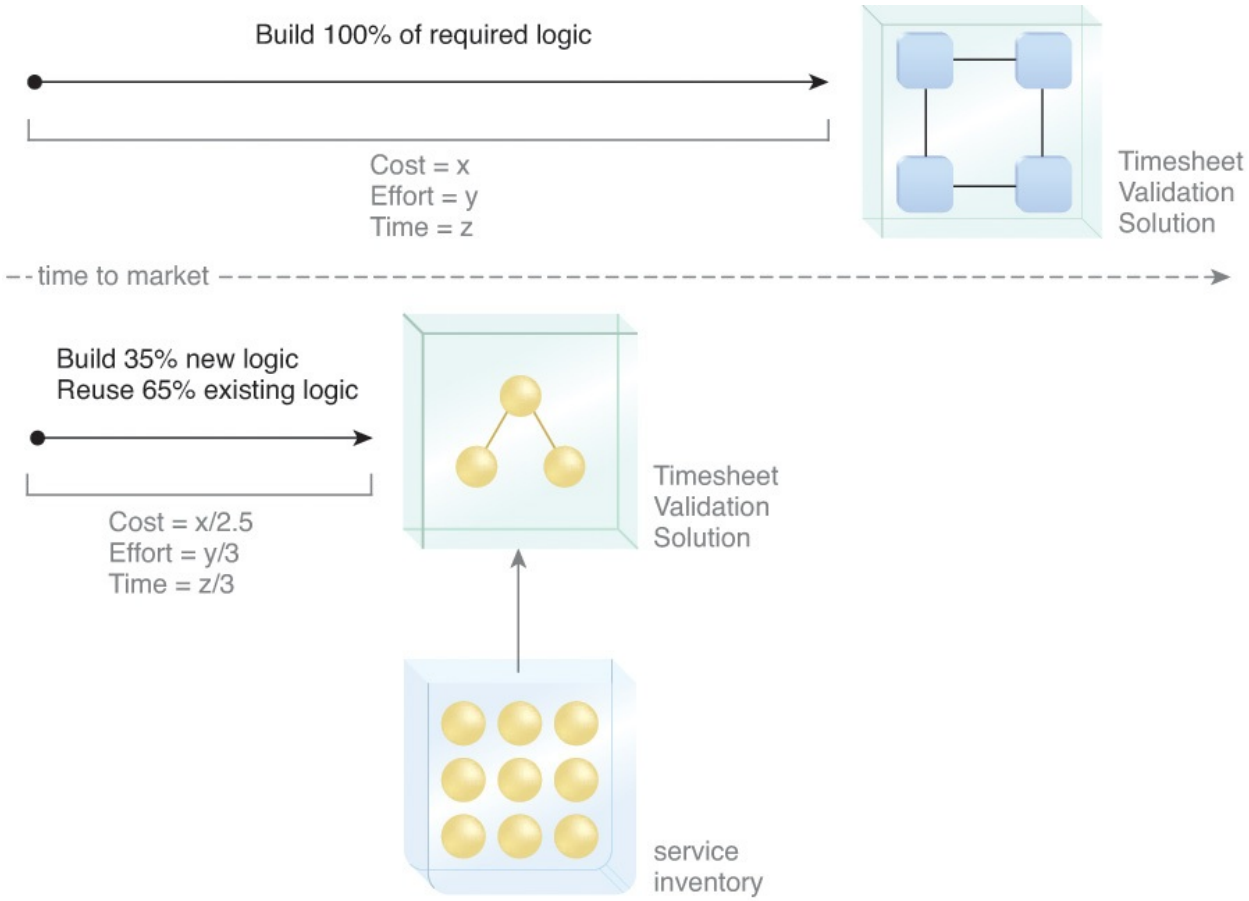


Figure 3.32 The delivery timeline is projected based on the percentage of "net new" solution logic that needs to be built. Though in this example only 35%

of new logic is required, the timeline is reduced by around 50% because significant effort is still required to incorporate existing, reusable services from the inventory.

The net result of this fundamental shift in project delivery is heightened responsiveness and reduced time to market potential, all of which translates into increased organizational agility.

Note

Organizational agility represents a target state that organizations work toward as they deliver services and populate service inventories. The organization benefits from increased responsiveness after a significant amount of services is in place. The processes required to model and design those services require more upfront cost and effort than building the corresponding quantity of solution logic using traditional project delivery approaches.

It is therefore important to acknowledge that service-orientation has a strategic focus that intends to establish a highly agile enterprise. This is different from agile development approaches that have more of a tactical focus.

Reduced IT Burden

Consistently applying service-orientation results in an IT enterprise with reduced waste and redundancy, reduced size and operational cost (Figure 3.33), and reduced overhead associated with its governance and evolution. Such an enterprise can benefit an organization through dramatic increases in efficiency and cost-effectiveness.

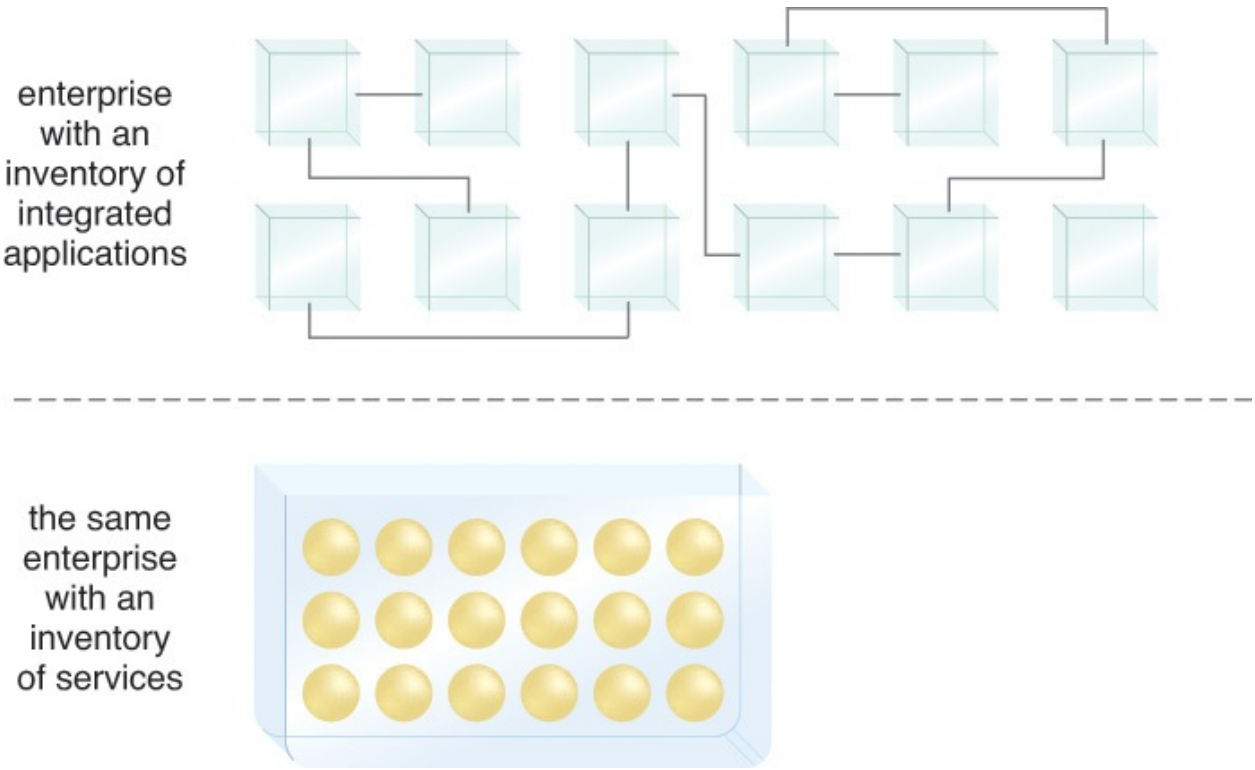


Figure 3.33 If you were to take a typical automated enterprise and redevelop it entirely with custom, normalized services, its overall size would shrink considerably, resulting in a reduced operational scope.

In essence, the attainment of the previously described goals can create a leaner, more agile IT department, one that is less of a burden on the organization and more of an enabling contributor to its strategic goals.

In summary, the consistent application of service-orientation design principles to individual services that eventually comprise a greater service inventory is the core requirement to achieving the goals and benefits of service-oriented computing (Figure 3.34).

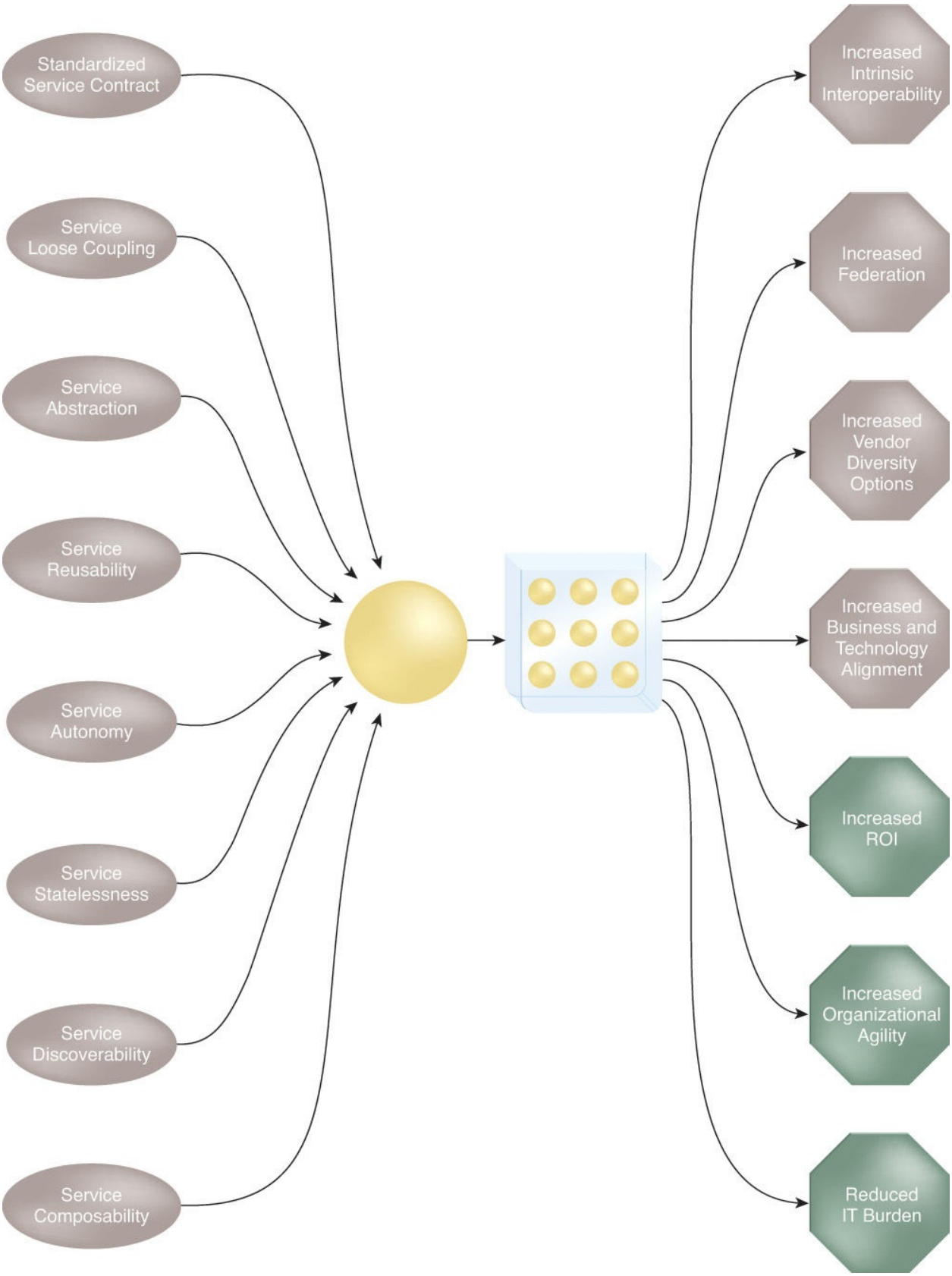


Figure 3.34 The repeated application of service-orientation principles to

services that are delivered as part of a collection leads to a target state based on the manifestation of the strategic goals associated with service-oriented computing.

3.5 Four Pillars of Service-Orientation

As previously explained, service-orientation provides us with a well-defined method for shaping software programs into units of service-oriented logic that we can legitimately refer to as services. Each such service that we deliver takes us a step closer to achieving the desired target state represented by the aforementioned strategic goals and benefits.

Proven practices, patterns, principles, and technologies exist in support of service-orientation. However, because of the distinctly strategic nature of the target state that service-orientation aims to establish, there is a set of fundamental critical success factors that act as common prerequisites for its successful adoption. These critical success factors are referred to as *pillars* because they collectively establish a sound and healthy foundation upon which to build, deploy, and govern services.

The four pillars of service-orientation are

- *Teamwork* – Cross-project teams and cooperation are required.
- *Education* – Team members must communicate and cooperate based on common knowledge and understanding.
- *Discipline* – Team members must apply their common knowledge consistently.
- *Balanced Scope* – The extent to which the required levels of Teamwork, Education, and Discipline need to be realized is represented by a meaningful yet manageable scope.

The existence of these four pillars is considered essential to any SOA initiative. The absence of any one of these pillars to a significant extent introduces a major risk factor. If such an absence is identified in the early planning stages, it can warrant not proceeding with the project until it has been addressed—or the project's scope has been reduced.

Teamwork

Whereas traditional silo-based applications require cooperation among members of individual project teams, the delivery of services and service-oriented solutions requires cooperation across multiple project teams. The scope of the required teamwork is noticeably larger and can introduce new dynamics, new project roles, and the need to forge and maintain new relationships among individuals and departments. Those on the overall SOA team need to trust and rely on each other; otherwise the team will fail.

Education

A key factor to realizing the reliability and trust required by SOA team members is to ensure that they use a common communications framework based on common vocabulary, definitions, concepts, methods, and a common understanding of the target state the team is collectively working to attain. To achieve this common understanding requires common education, not just in general topics pertaining to service-orientation, SOA, and service technologies, but also in specific principles, patterns, and practices, as well as established standards, policies, and methodology specific to the organization.

Combining the pillars of teamwork and education establishes a foundation of knowledge and an understanding of how to use that knowledge among members of the SOA team. The resulting clarity eliminates many of the common risks that have traditionally plagued SOA projects.

Discipline

A critical success factor for any SOA initiative is consistency in how knowledge and practices among a cooperative team are used and applied. To be successful as a whole, team members must therefore be disciplined in how they apply their knowledge and in how they carry out their respective roles. Required measures of discipline are commonly expressed in methodology, modeling, and design standards, as well as governance precepts. Even with the best intentions, an educated and cooperative team will fail without discipline.

Balanced Scope

So far we've established that we need:

- cooperative teams that have…
- a common understanding and education pertaining to industry and enterprise-specific knowledge areas and that…
- we need to consistently cooperate as a team, apply our understanding, and follow a common methodology and standards in a disciplined manner.

In some IT enterprises, especially those with a long history of building silo-based applications, achieving these qualities can be challenging. Cultural, political, and various other forms of organizational issues can arise to make it difficult to attain the necessary organizational changes required by these three pillars. How then can they be realistically achieved? It all comes down to defining a balanced scope of adoption.

The scope of adoption needs to be meaningfully cross-silo, while also realistically manageable. This requires the definition of a balanced scope of adoption of service-orientation.

Note

The concept of a balanced scope corresponds directly to the following guideline in the SOA Manifesto:

"The scope of SOA adoption can vary. Keep efforts manageable and

within meaningful boundaries.”

See Appendix D for the complete SOA Manifesto and the Annotated SOA Manifesto.

Once a balanced scope of adoption has been defined, this scope determines the extent to which the other three pillars need to be established. Conversely, the extent to which you can realize the other three pillars will influence how you determine the scope (Figure 3.35).

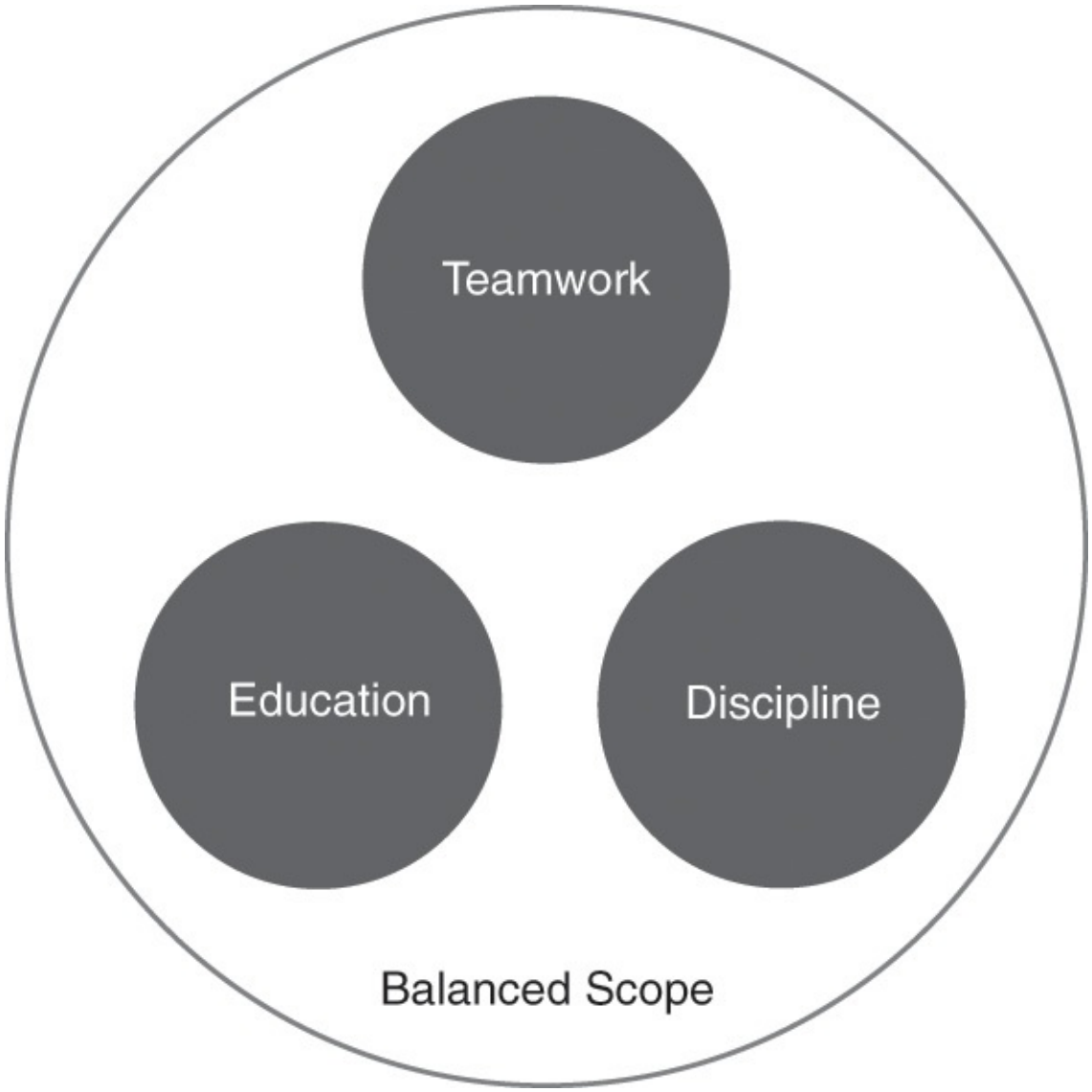


Figure 3.35 The Balanced Scope pillar encompasses and sets the scope at which the other three pillars are applied for a given adoption effort.

Common factors involved in determining a balanced scope include:

- Cultural obstacles
- Authority structures
- Geography
- Business domain alignment
- Available stakeholder support and funding
- Available IT resources

A single organization can choose one or more balanced adoption scopes (Figure 3.36). Having multiple scopes results in a domain-based approach to adoption. Each domain establishes a boundary for an inventory of services. Among domains, adoption of service-orientation and the delivery of services can occur independently. This does not result in application silos; it establishes meaningful service domains (also known as “continents of services”) within the IT enterprise.

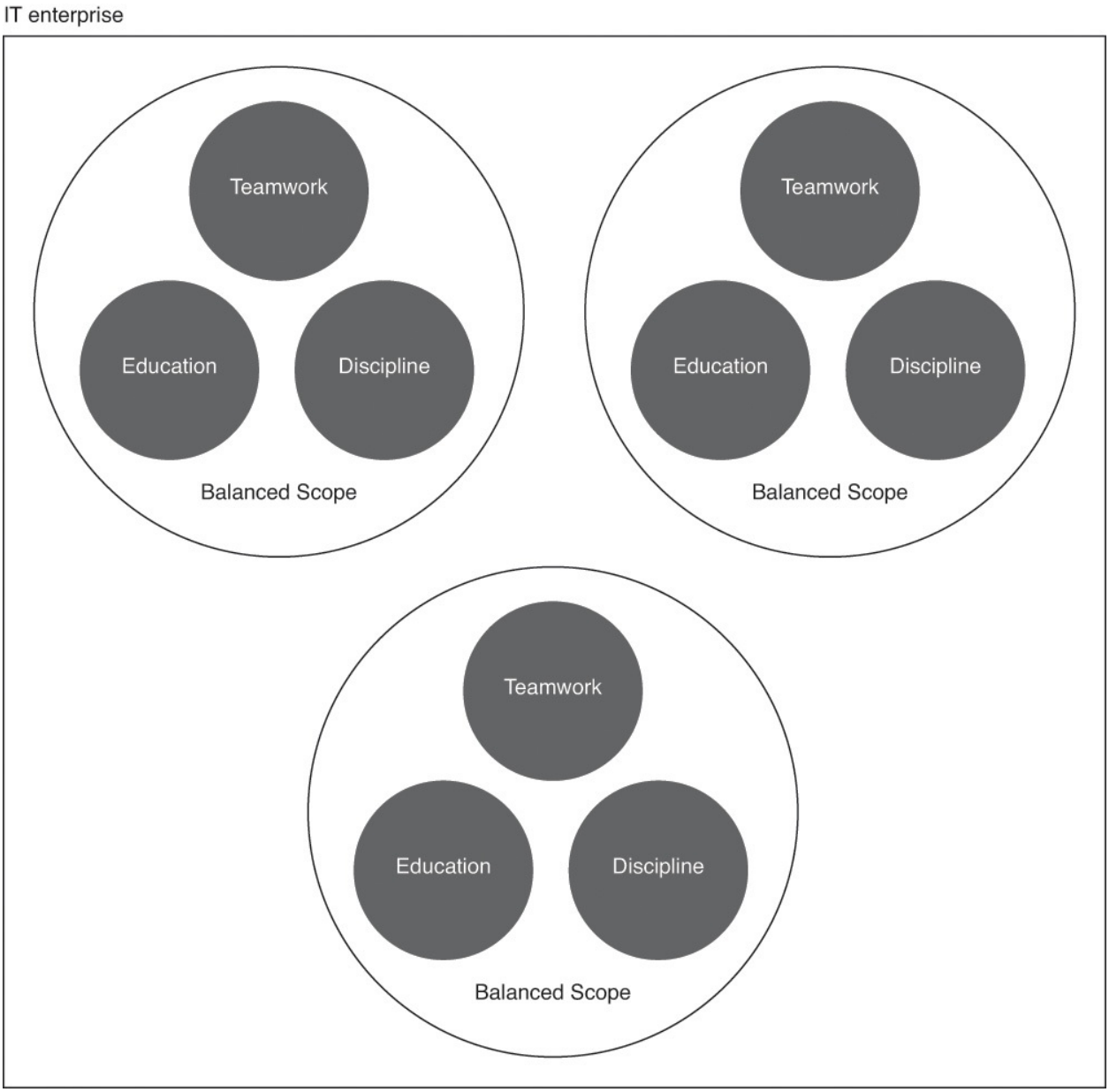


Figure 3.36 Multiple balanced scopes can exist within the same IT enterprise. Each represents a separate domain service inventory that is independently standardized, owned, and governed.

SOA Patterns

The domain service inventory originated with the Domain Inventory [338] pattern, which is an alternative to the Enterprise Inventory [340] pattern.

Chapter 4. Understanding SOA

The focus of this chapter is to establish the link between service-orientation and

technology architecture, establish distinct SOA characteristics and types, and raise key project delivery considerations.

Note

Several of the upcoming sections make reference to clouds and cloud computing in general. If you are new to cloud computing, you can find introductory content at www.whatiscloud.com and cloud computing patterns at www.cloudpatterns.org. More comprehensive coverage is provided in the *Cloud Computing: Concepts, Technology & Architecture* and *Cloud Computing Design Patterns* titles that are part of the *Prentice Hall Service Technology Series from Thomas Erl*.

Introduction to SOA

Let's briefly recap some of the topics covered in Chapter 3 to clearly establish how they relate to each other and how they specifically lead to a definition of SOA:

- There is a set of strategic goals associated with service-oriented computing.
- These goals represent a specific target state.
- Service-orientation is the paradigm that provides a proven method for achieving this target state.
- When we apply service-orientation to the design of software, we build units of logic called "services."
- Service-oriented solutions are comprised of one or more services.

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

In other words:

- To build successful service-oriented solutions, we need a distributed technology architecture with specific characteristics.

- These characteristics distinguish the technology architecture as being service-oriented. This is SOA.

Service-orientation is fundamentally about attaining the specific target state we established toward the end of Chapter 3. It asks that we take extra design considerations into account with everything we build so that all the moving parts of a given service-oriented solution support the realization of this state and foster its growth and evolution. These design considerations carry over into the supporting technology architecture, which must have a distinct set of characteristics that enable the target state and inherently accommodate ongoing change within that target environment.

4.1 The Four Characteristics of SOA

Service-oriented technology architecture must have certain properties that fulfill the fundamental requirements for an automation solution comprised of services to which service-orientation design principles have been applied. These four characteristics further help distinguish SOA from other architectural models.

Note

As we explore each of these characteristics individually, keep in mind that in real-world implementations the extent to which these characteristics can be attained will likely vary.

Business-Driven

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

This gradual separation of business and technology results in a technology architecture with diminishing potential to fulfill business requirements and one that is increasingly difficult to adapt to changing business needs (Figure 4.1).

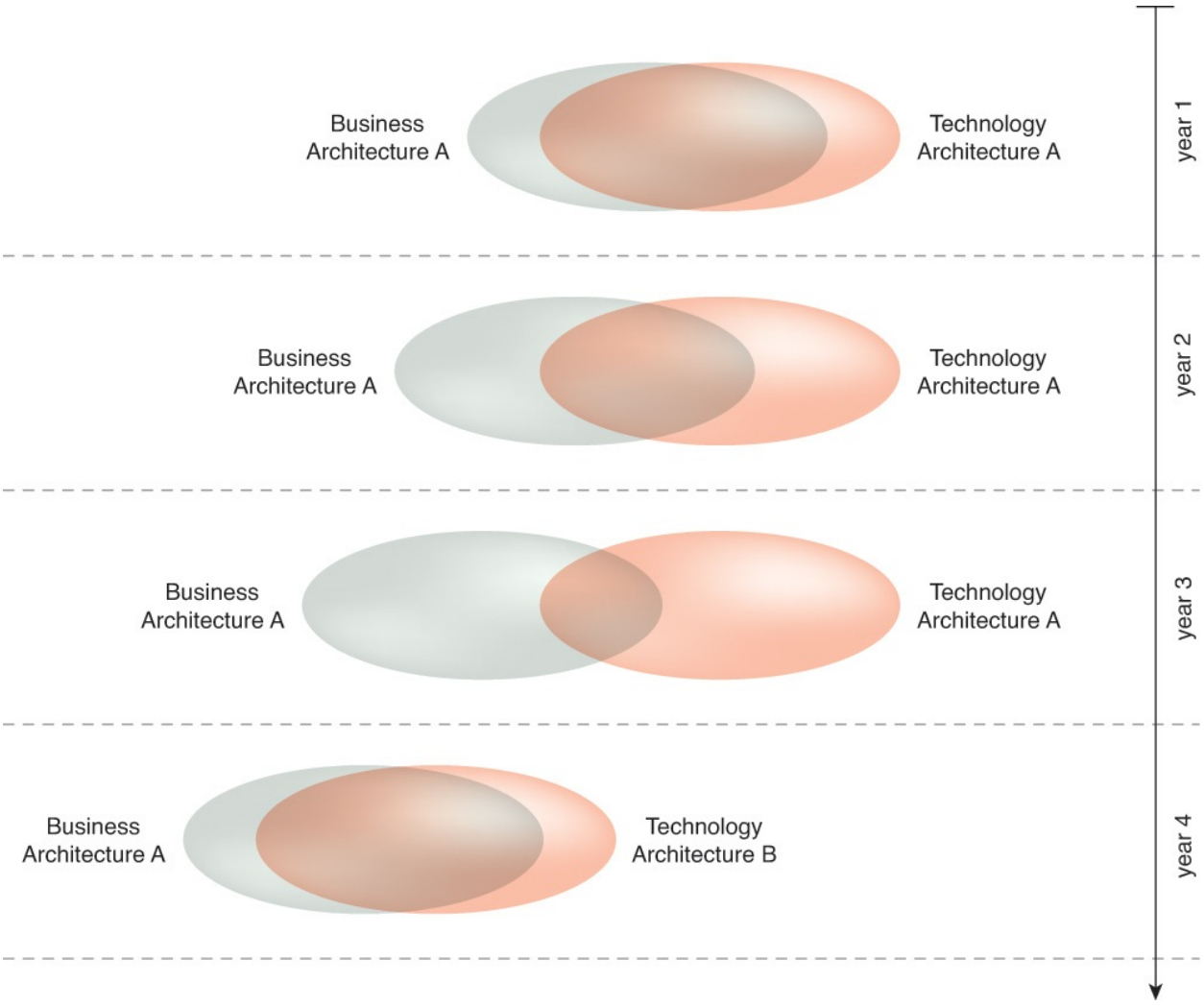


Figure 4.1 A technology architecture (A) is often delivered in alignment with the current state of a business but can be incapable of changing in alignment with how the business evolves. As business and technology architectures become increasingly out of sync, business requirement fulfillment decreases, often to the point that a whole new technology architecture (B) is needed, which effectively resets this cycle.

When a technology architecture is business-driven, the overarching business vision, goals, and requirements are positioned as the basis for and the primary influence of the architectural model. This maximizes the potential alignment of technology and business and allows for a technology architecture that can evolve in tandem with the organization as a whole (Figure 4.2). The result is a continual increase in the value and lifespan of the architecture.

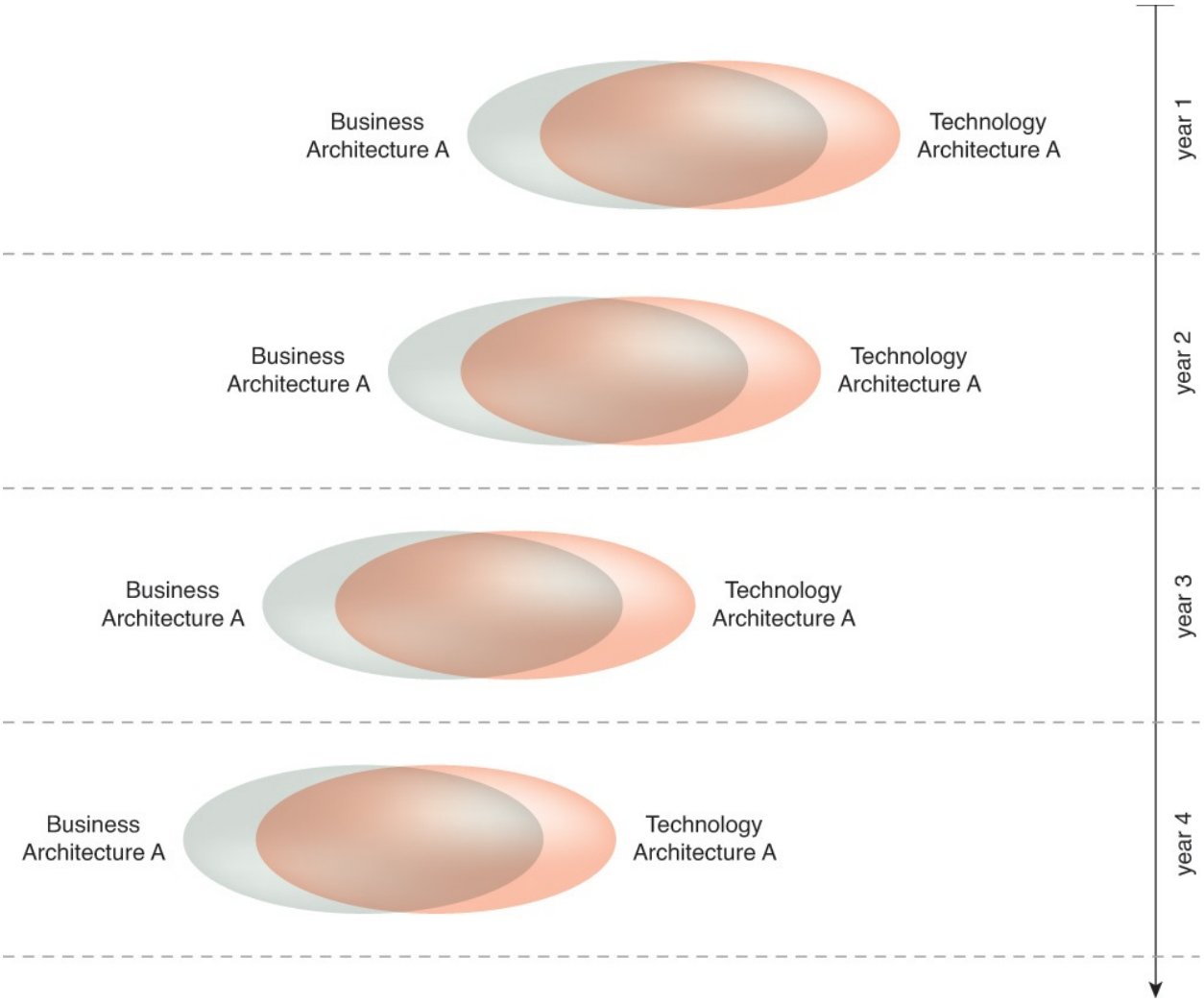


Figure 4.2 By defining a strategic, business-centric scope to the technology architecture, it can be kept in constant sync with how the business evolves over time.

Vendor-Neutral

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

An inhibitive technology architecture is unable to evolve and expand in response to changing automation requirements, which can result in the architecture having a limited lifespan after which it needs to be replaced to remain effective (Figure 4.3).

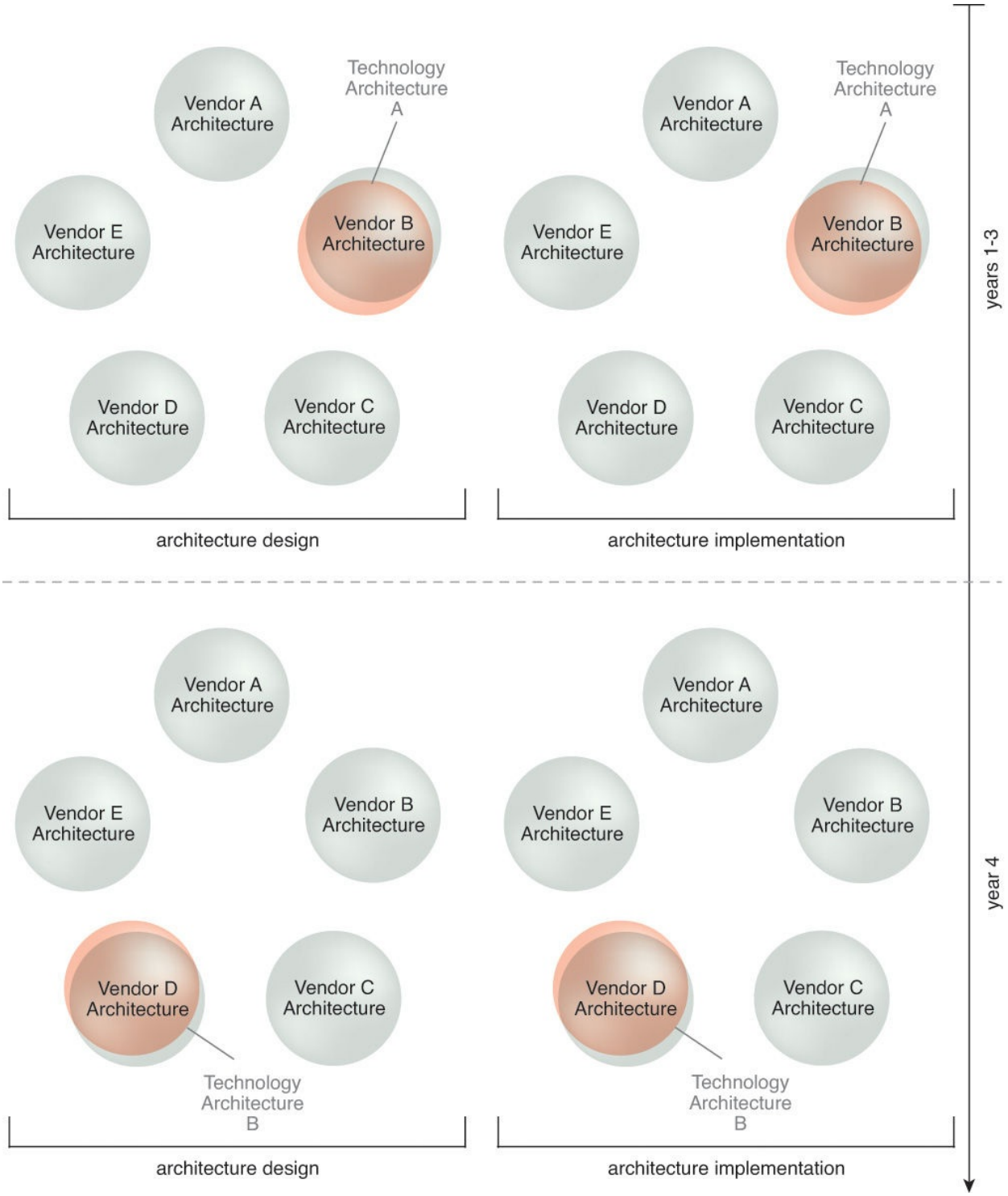


Figure 4.3 Vendor-centric technology architectures are often bound to corresponding vendor platform roadmaps. This can reduce opportunities to leverage technology innovations provided by other vendor platforms and can result in the need to eventually replace the architecture entirely with a new vendor implementation (which starts the cycle over again).

It is in the best interest of an organization to base the design of a service-oriented architecture on a model that is in alignment with the primary SOA vendor platforms, yet neutral to all of them. A vendor-neutral architectural model can be derived from a vendor-neutral design paradigm used to build the solution logic the architecture will be responsible for supporting (Figure 4.4). The service-orientation paradigm provides such an approach, in that it is derived from and applicable to real-world technology platforms while remaining neutral to them.

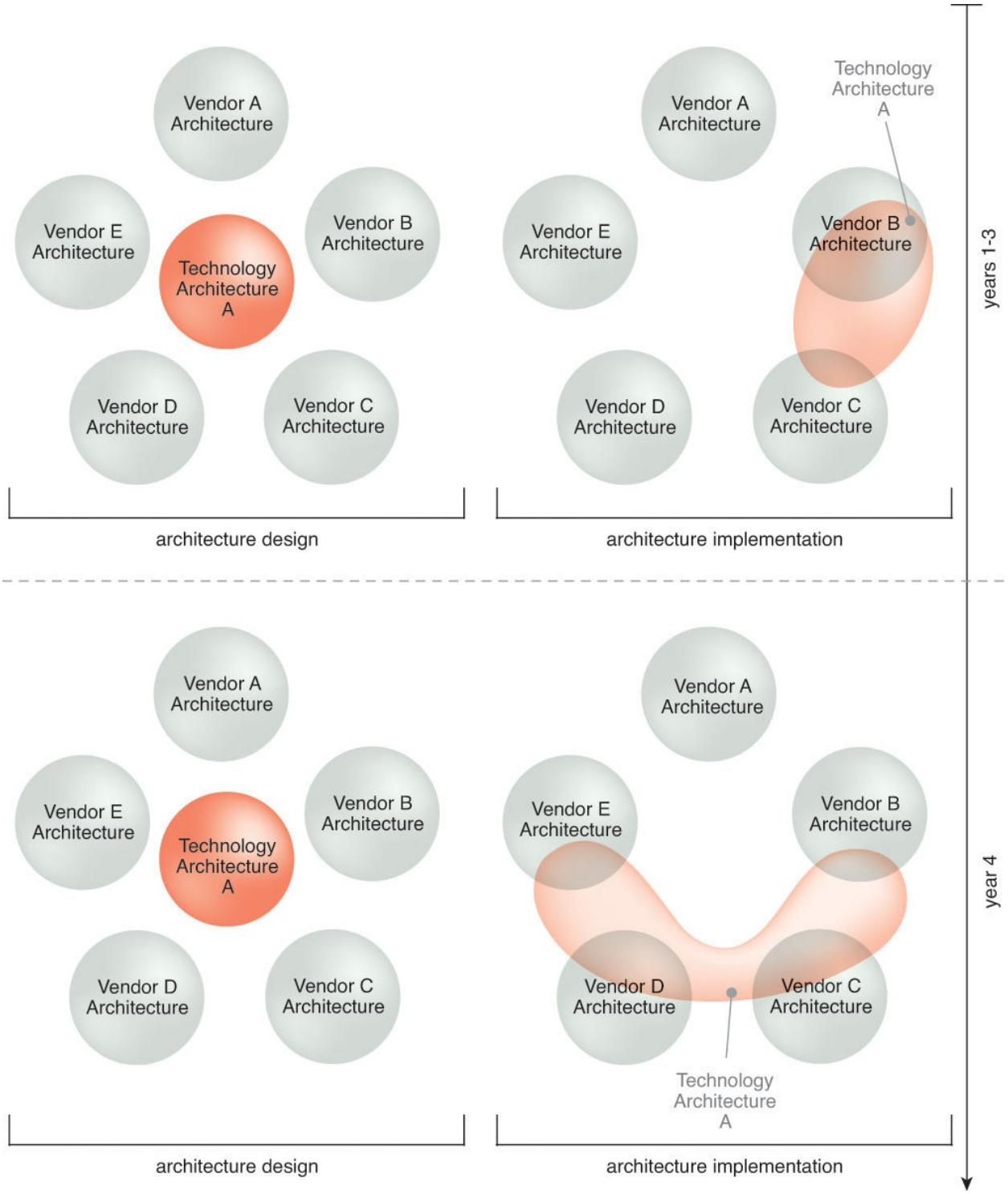


Figure 4.4 If the architectural model is designed to be and remain neutral to vendor platforms, it maintains the freedom to diversify its implementation by leveraging multiple vendor technology innovations. This increases the longevity of the architecture as it is allowed to augment and evolve in response to changing requirements.

Note

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

Enterprise-Centric

The fact that service-oriented solutions are based on a distributed architecture doesn't mean that there still isn't the constant danger of creating new silos within an enterprise when building poorly designed services, as illustrated in Figure 4.5.

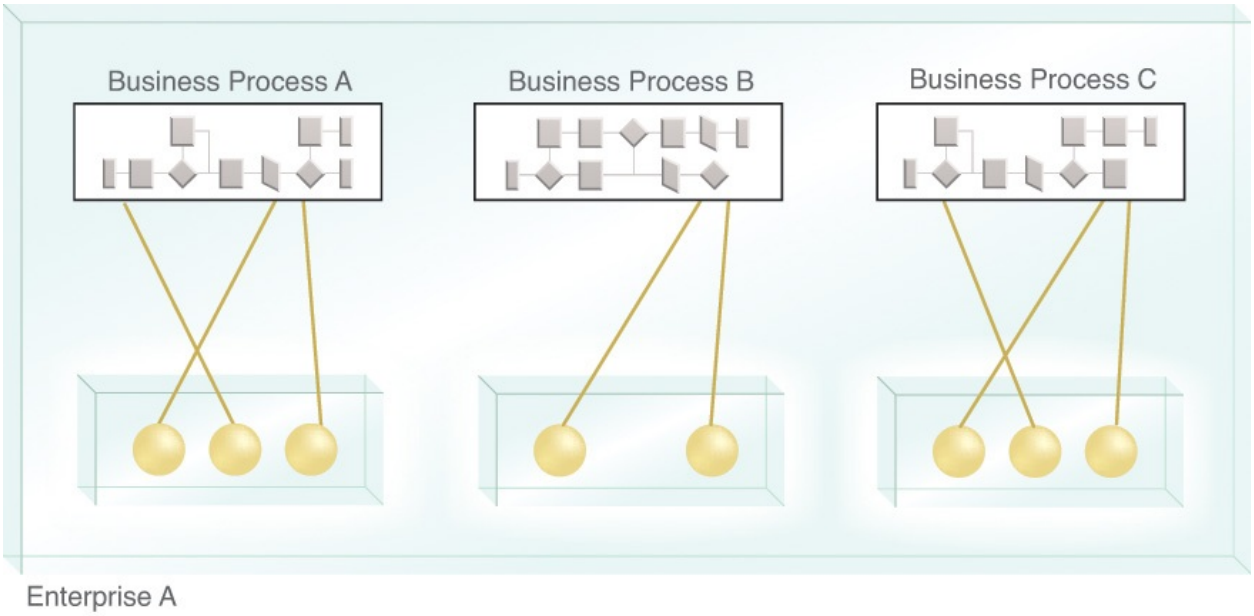


Figure 4.5 Single-purpose services delivered to automate specific business processes can end up establishing silos within the enterprise.

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

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

Essentially, the body of logic is classified as a resource of the enterprise. This does not necessarily make it an enterprise-*wide* resource or one that must be

used throughout an entire technical environment. An enterprise resource is simply logic positioned as an IT asset; an extension of the enterprise that does not belong solely to any one application or solution.

SOA Patterns

As established in the Service Encapsulation [359] pattern, an enterprise resource essentially embodies the fundamental characteristics of service logic.

To leverage services as enterprise resources, the underlying technology architecture must establish a model that is natively based on the assumption that software programs delivered as services will be shared by other parts of the enterprise or will be part of larger solutions that include shared services. This baseline requirement places an emphasis on standardizing parts of the architecture so that service reuse and interoperability can be continually fostered (Figure 4.6).

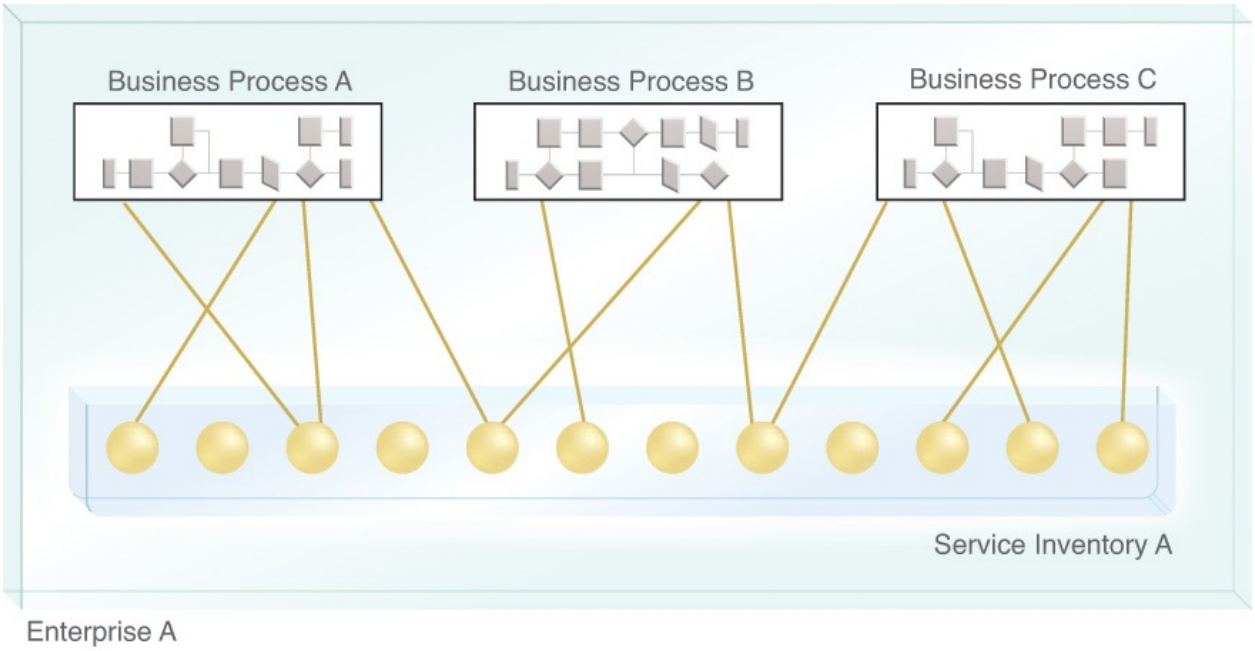


Figure 4.6 When services are positioned as enterprise resources, they no longer create or reside in silos. Instead they are made available to a broader scope of utilization by being part of a service inventory.

Composition-Centric

More so than in previous distributed computing paradigms, service-orientation places an emphasis on designing software programs as not just reusable

resources, but as flexible resources that can be plugged into different aggregate structures for a variety of service-oriented solutions.

To accomplish this, services must be composable. As advocated by the Service Composability (302) principle, this means that services must be capable of being pulled into a variety of composition designs, regardless of whether or not they are initially required to participate in a composition when they are first delivered (Figure 4.7).

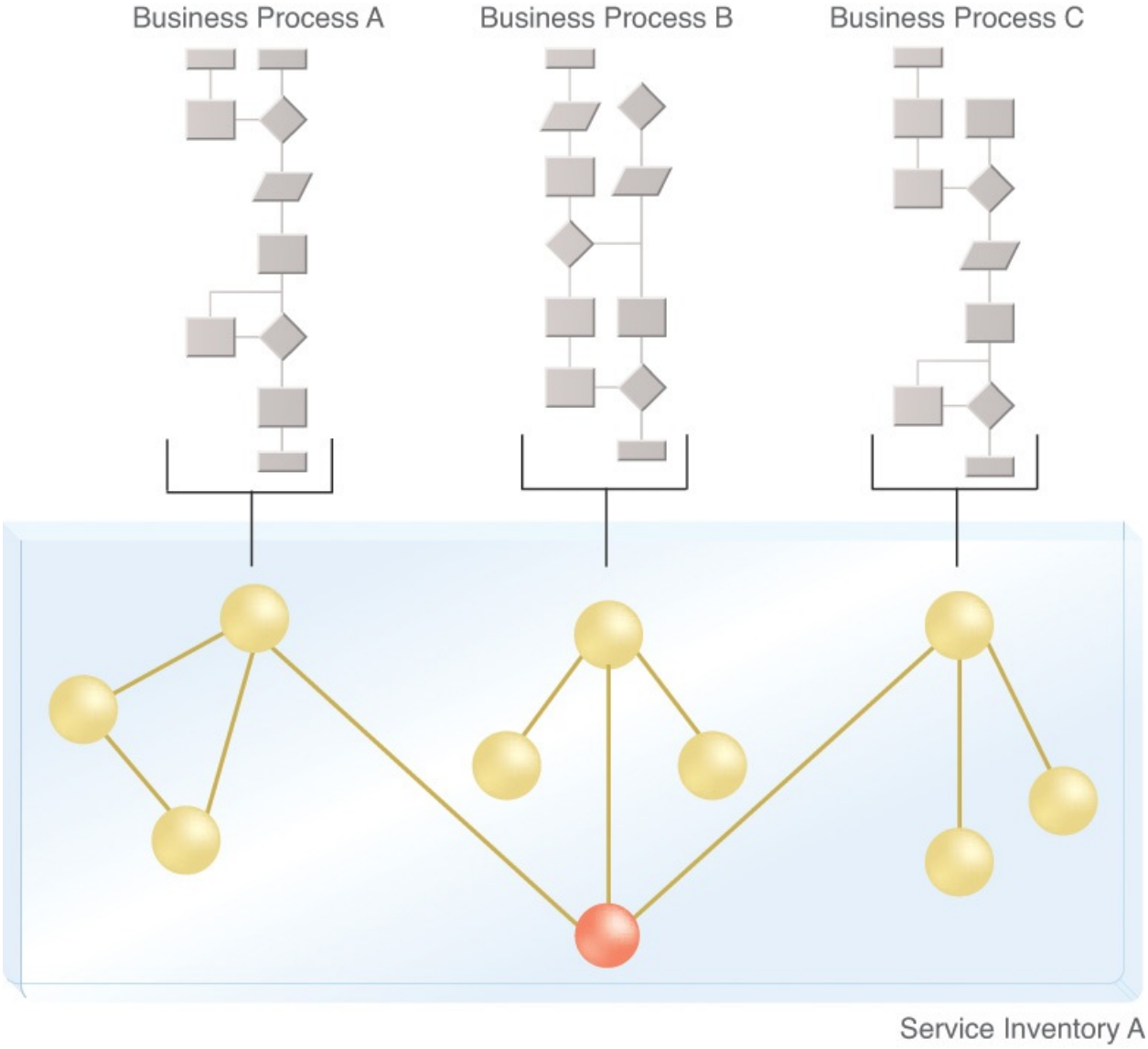


Figure 4.7 Services within the same service inventory are composed into different configurations. The highlighted service is reused by multiple compositions to automate different business processes.

To support native composability, the underlying technology architecture must be prepared to enable a range of simple and complex composition designs.

Architectural extensions (and related infrastructure extensions) pertaining to scalability, reliability, and runtime data exchange processing and integrity are essential to support this key characteristic.

Design Priorities

A valuable perspective of how service-orientation relates to SOA and of how the formalization of this relationship results in a set of design priorities was provided by the publication of the "SOA Manifesto." Have a look at the following excerpt:

> Service orientation is a paradigm that frames what you do. Service-oriented architecture (SOA) is a type of architecture that results from applying service orientation.
>
> We have been applying service orientation to help organizations consistently deliver sustainable business value, with increased agility and cost effectiveness, in line with changing business needs.
>
> Through our work we have come to prioritize:
>
> Business value over technical strategy
>
> Strategic goals over project-specific benefits
>
> Intrinsic interoperability over custom integration
>
> Shared services over specific-purpose implementations
>
> Flexibility over optimization
>
> Evolutionary refinement over pursuit of initial perfection
>
> That is, while we value the items on the right, we value the items on the left more.

It is evident how these design priorities are directly supported by the service-orientation design paradigm and the service-oriented architectural model. This is further explored in the "Annotated SOA Manifesto" that was published at www.soa-manifesto.com and is also provided in Appendix D of this book.

4.2 The Four Common Types of SOA

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

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

To better understand the basic mechanics of SOA, we now need to study the common types of technology architectures that exist within a typical service-oriented environment:

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

SOA Patterns

Architecture types are closely related to SOA patterns. Note how each pattern profile table in Appendix C contains a field dedicated to showing related architectures.

The service-oriented enterprise architecture represents a parent architecture that encompasses all others. The environment and conventions established by this parent platform are carried over into the service inventory architecture implementations that may reside within a single enterprise environment. These inventories further introduce new and more specific architectural elements (such as runtime platforms and middleware) that then form the foundation of service and composition architectures implemented within an inventory's boundary.

As a result, a natural form of architectural inheritance is formed whereby more granular architecture implementations inherit elements from less granular ones (Figure 4.8). This relationship between architecture types is good to keep in mind as it can identify potential (positive and negative) dependencies that may exist.

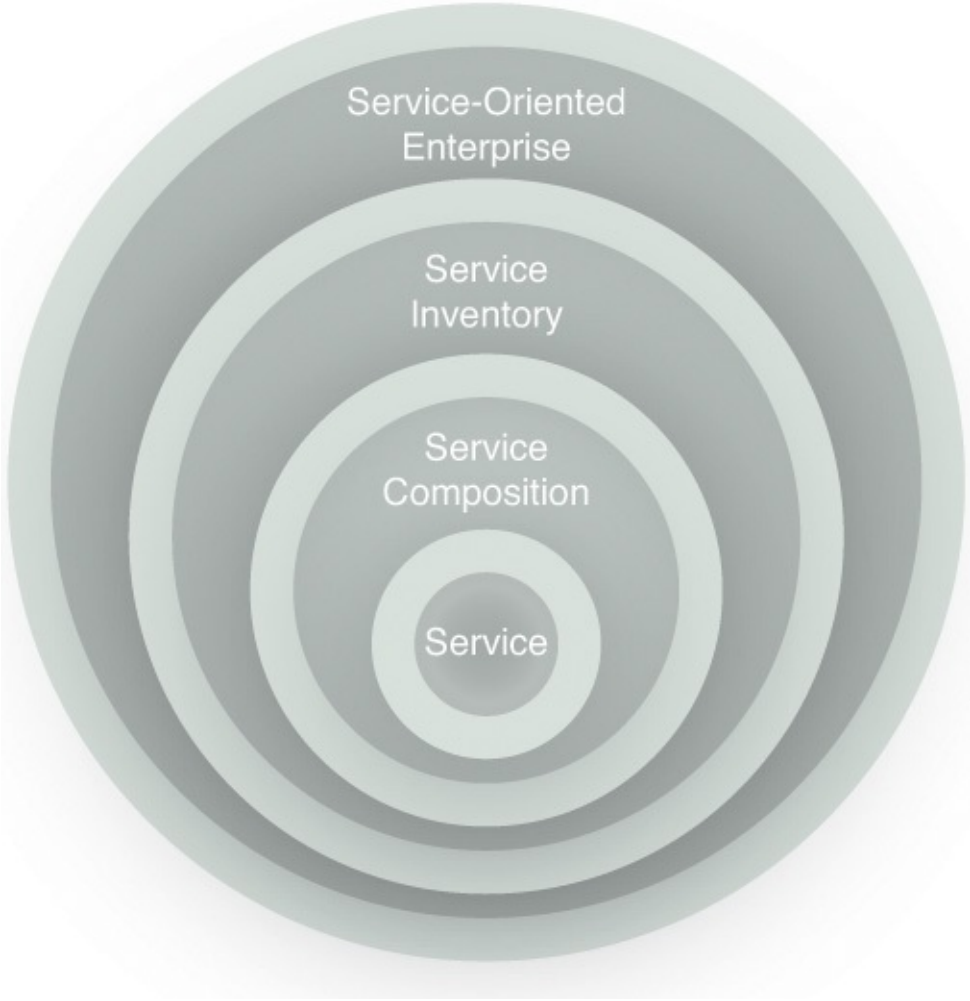


Figure 4.8 The layered SOA model establishes the four common SOA types: service architecture, service composition architecture, service inventory architecture, and service-oriented enterprise architecture.

The following section explores the architecture types individually and concludes by highlighting links between these characteristics and common SOA design priorities.

Service Architecture

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

other things, encompass multiple components (Figure 4.9).

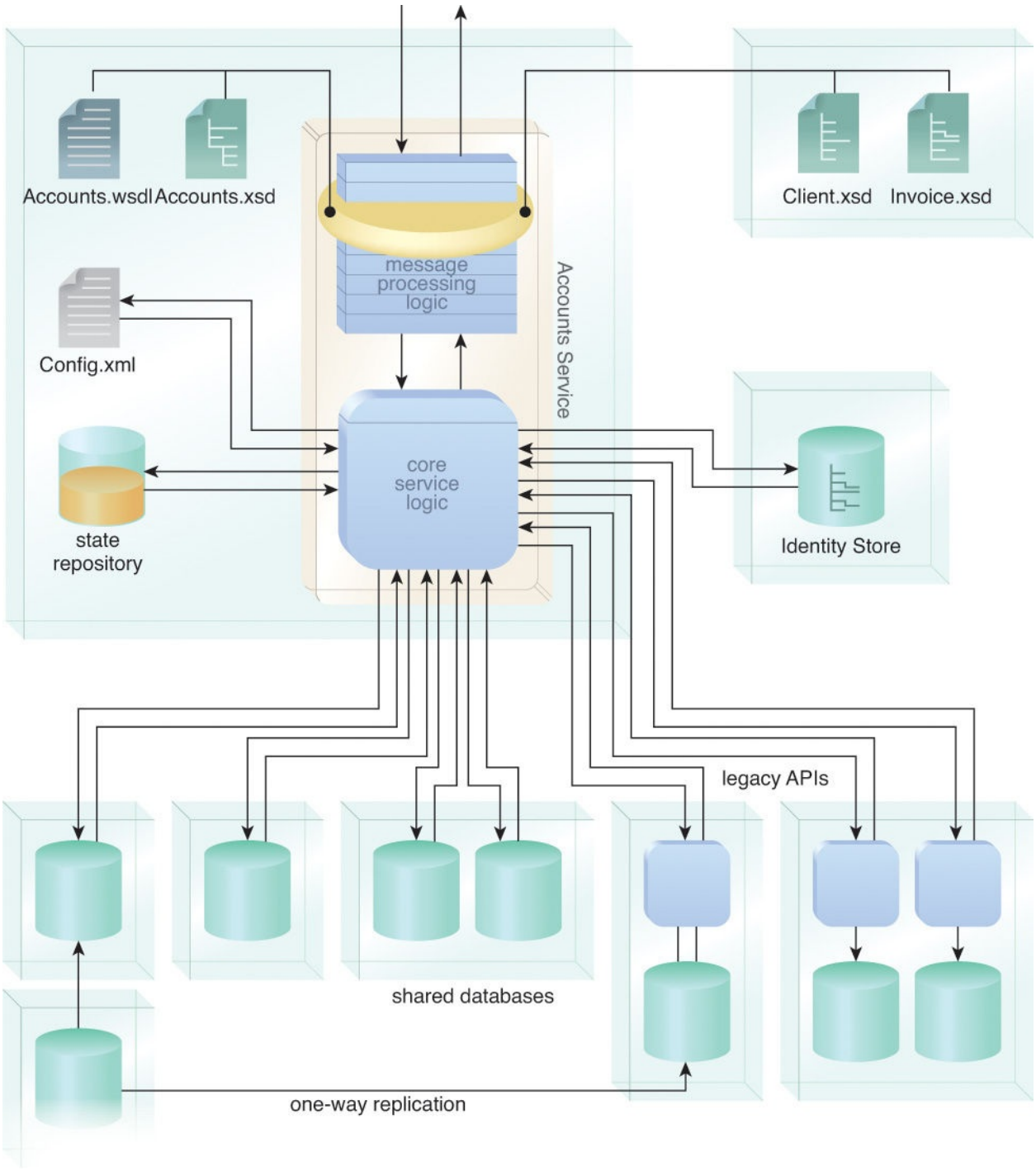


Figure 4.9 An example of a high-level service architecture view for the Accounts service, depicting the parts of the surrounding infrastructure utilized to fulfill the functional requirements of all capabilities. Additional views can be created to show only those architectural elements related to the processing of specific capabilities. Further detail, such as data flow and security requirements, would normally also be included.

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

Service architecture specifications are typically owned by service custodians and, in support of the Service Abstraction (294) design principle, their contents are often protected and hidden from other project team members (Figure 4.10).

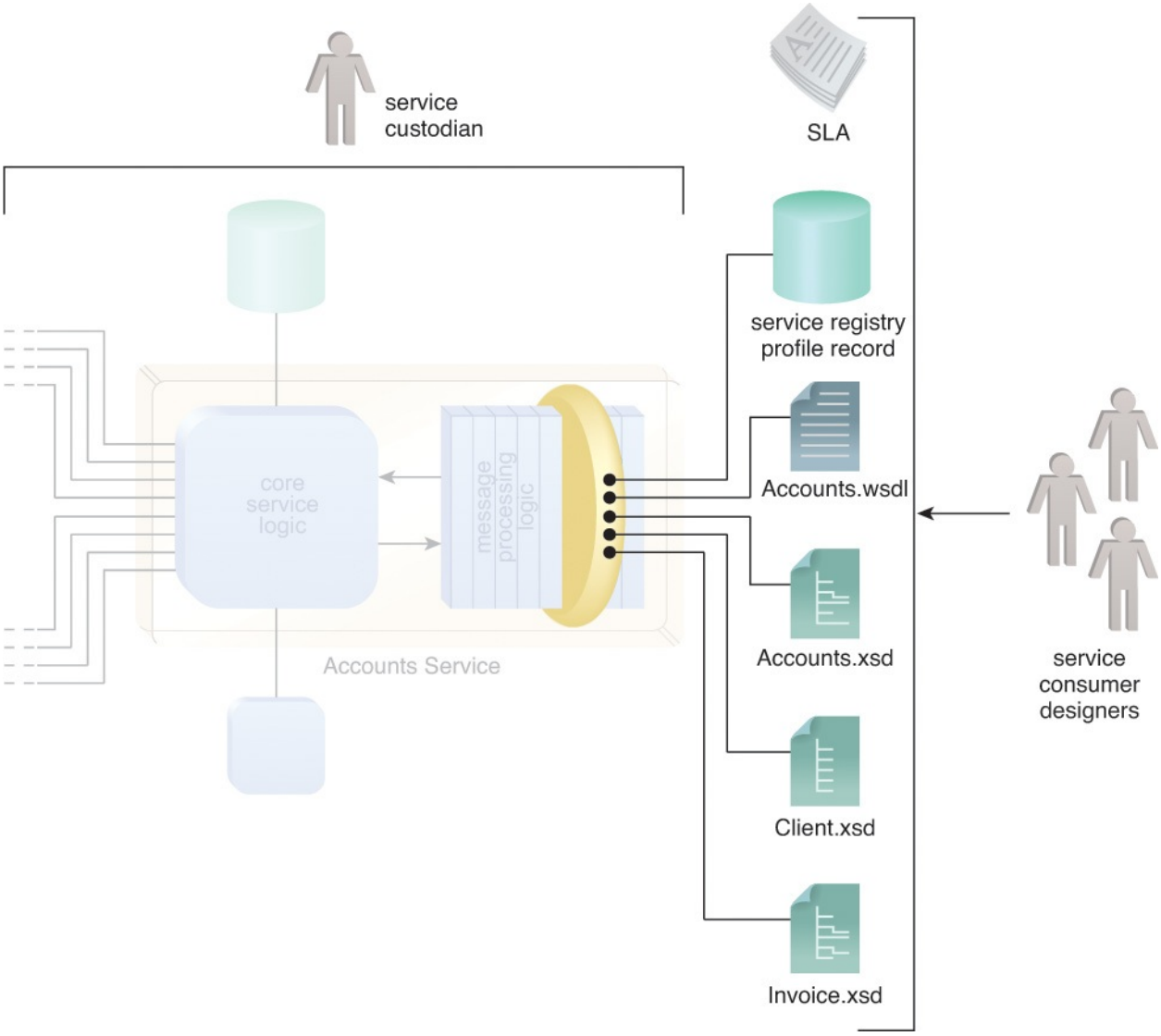


Figure 4.10 The custodian of the Accounts service intentionally limits access to architecture documentation. As a result, service consumer designers are only privy to published service contract documents.

The application of design standards and other service-orientation design principles further affects the depth and detail to which a service's technology

architecture may need to be defined (Figure 4.11). For example, implementation considerations raised by the Service Autonomy (297) and Service Statelessness (298) principles can require a service architecture to extend deeply into its surrounding infrastructure by defining exactly what physical environment it is deployed within, what resources it needs to access, what other parts of the enterprise may be accessing those same resources, and what extensions from the infrastructure it can use to defer or store data it is responsible for processing.

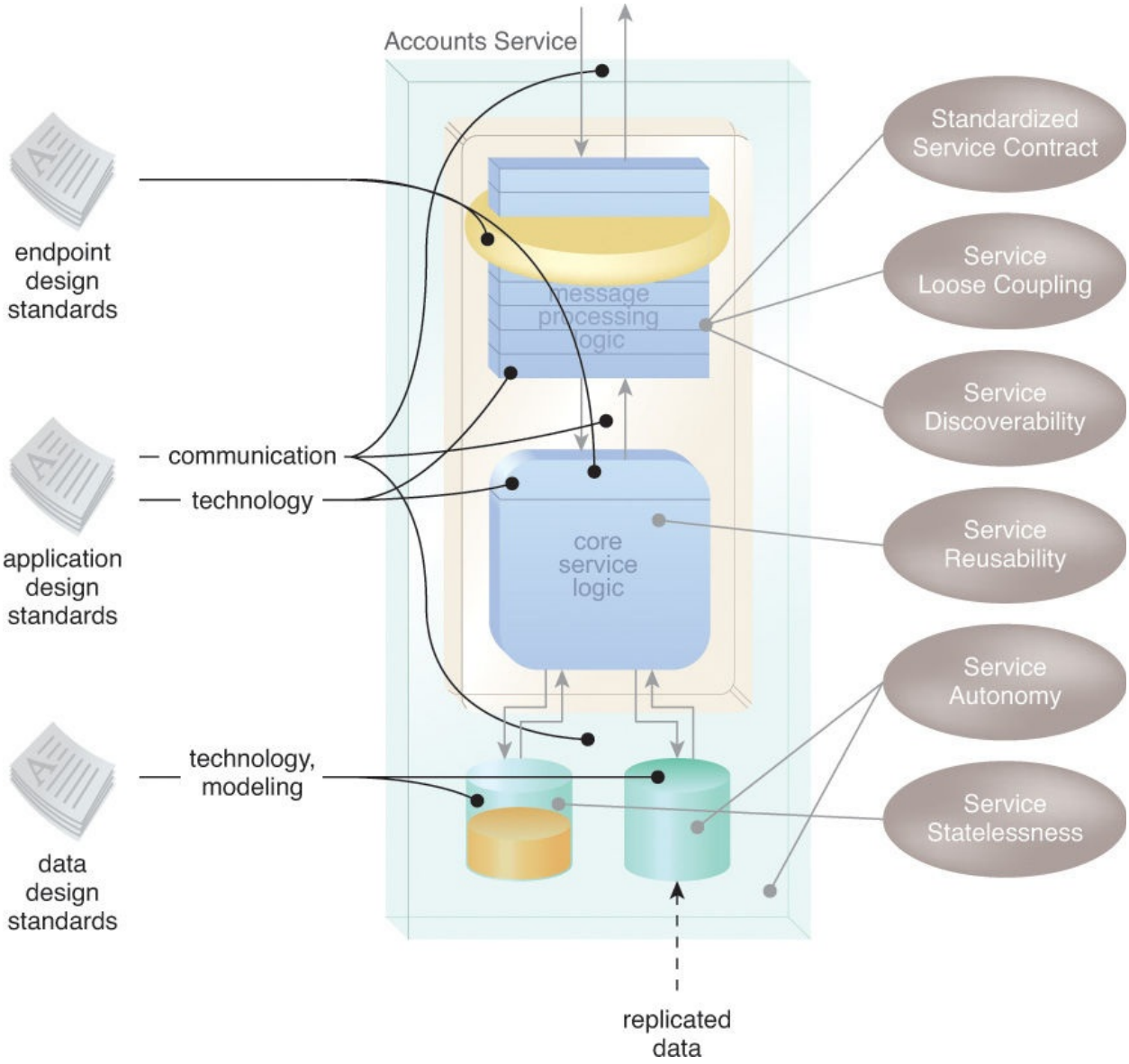


Figure 4.11 Custom design standards and service-orientation design principles are applied to establish a specific set of design characteristics within the Accounts service architecture.

A central part of a service architecture is typically its API. Following standard

service-oriented design processes, the service contract is generally the first part of a service to be physically delivered. The capabilities expressed by the contract further dictate the scope and nature of its underlying logic and the processing requirements that will need to be supported by its implementation (Figure 4.12).

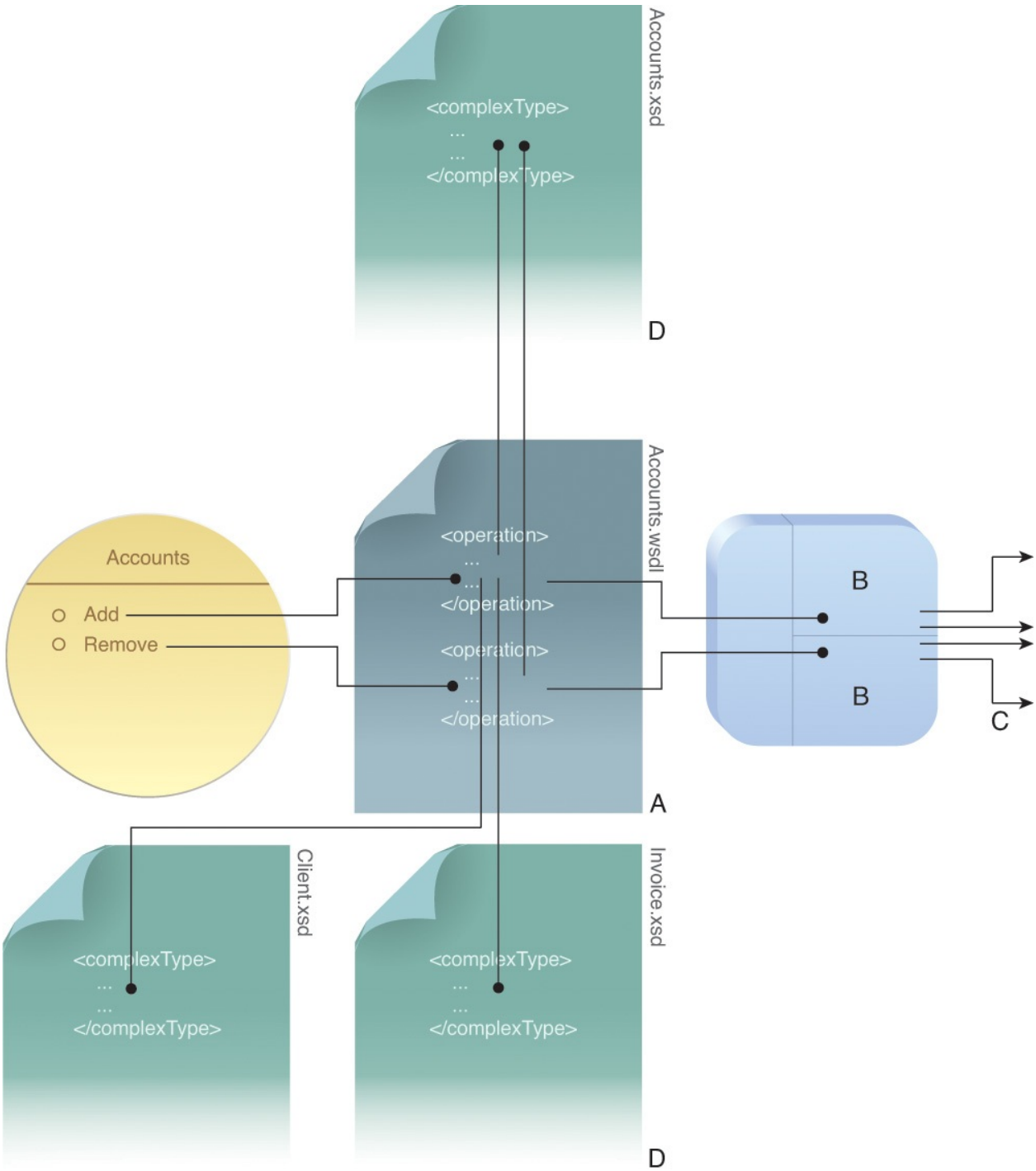


Figure 4.12 The service contract is a fundamental part of the Accounts service architecture. Its definition gives the service a public identity and helps express its functional scope. Specifically, the WSDL document (A) expresses operations that correspond to segments of functionality (B) within the underlying Accounts service logic. The logic, in turn, accesses other resources in the enterprise to carry out those functions (C). To accomplish this, the WSDL document provides data exchange definitions via input and output

message types established in separate XML schema documents (D).

This is why some consideration is given to implementation during the service modeling phase. The details documented during this analysis stage are carried forth into design, and much of this information can make its way into the official architecture definition.

Note

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

Another infrastructure-related aspect of service design that may be part of a service architecture is any dependencies the service may have on service *agents* —event-driven intermediary programs capable of transparently intercepting and processing messages sent to or from a service.

SOA Patterns

Service agents can be custom-developed or may be provided by the underlying runtime environment, as per the Service Agent [357] pattern.

Within a service architecture the specific agent programs may be identified along with runtime information as to how message contents are processed or even altered by agent involvement. Service agents may themselves also have architecture specifications that can be referenced by the service architecture (Figure 4.13).

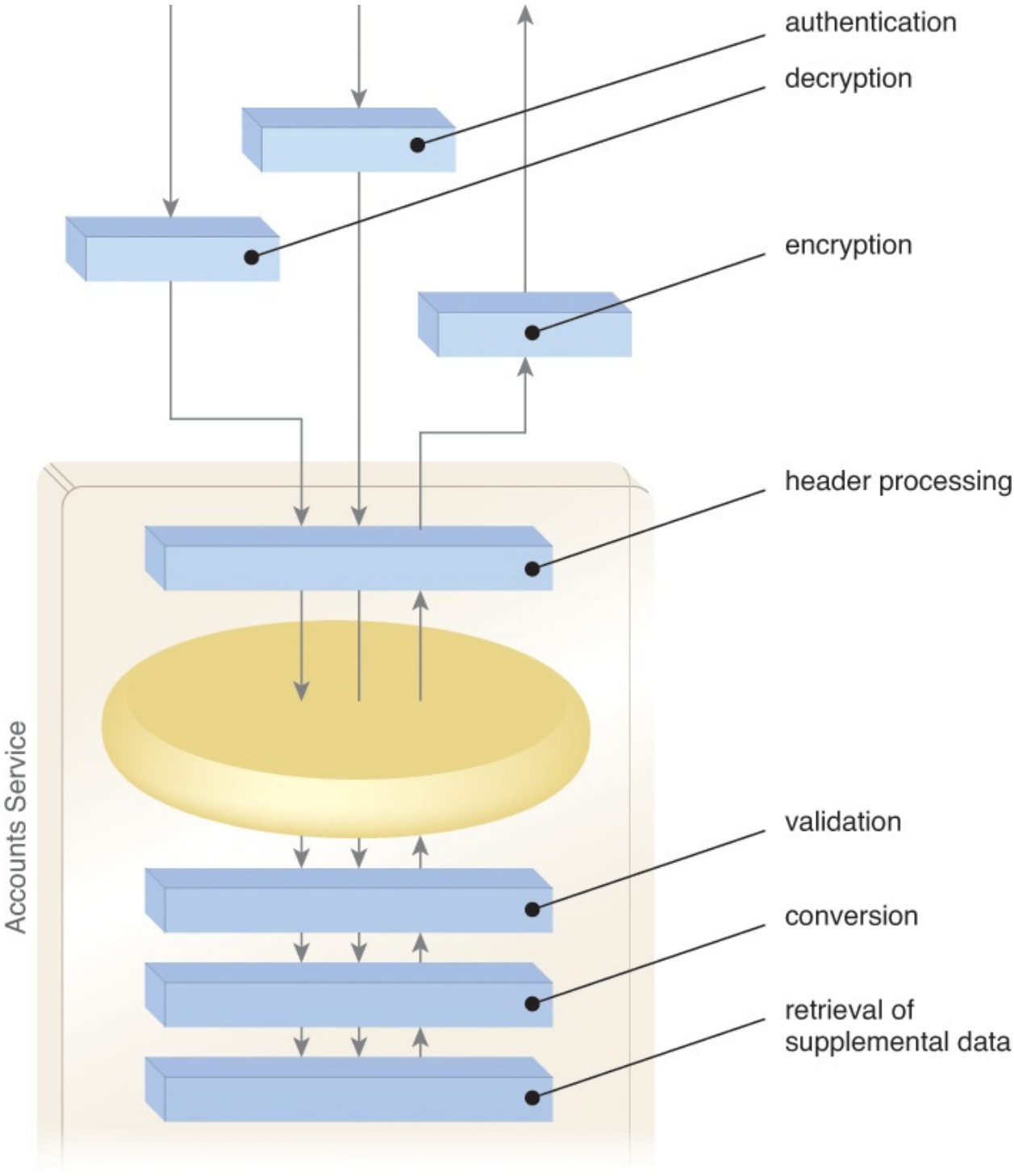


Figure 4.13 A variety of service agents are part of the Accounts service architecture. Some perform general processing of all data whereas others are specific to input or output data flow.

A key aspect of any service architecture is the fact that the functionality offered

by a service resides within one or more individual capabilities. This often requires the architecture definition itself to be taken to the capability level.

Each service capability encapsulates its own piece of logic. Some of this logic may be custom-developed for the service, whereas other capabilities may need to access one or more legacy resources. Therefore, individual capabilities end up with their own, individual designs that may need to be so detailed that they are documented as separate “capability architectures.” However, all relate back to the parent service architecture.

Service Composition Architecture

The fundamental purpose of delivering a series of independent services is so they can be combined into *service compositions*, fully functional solutions capable of automating larger, more complex business tasks (Figure 4.14).

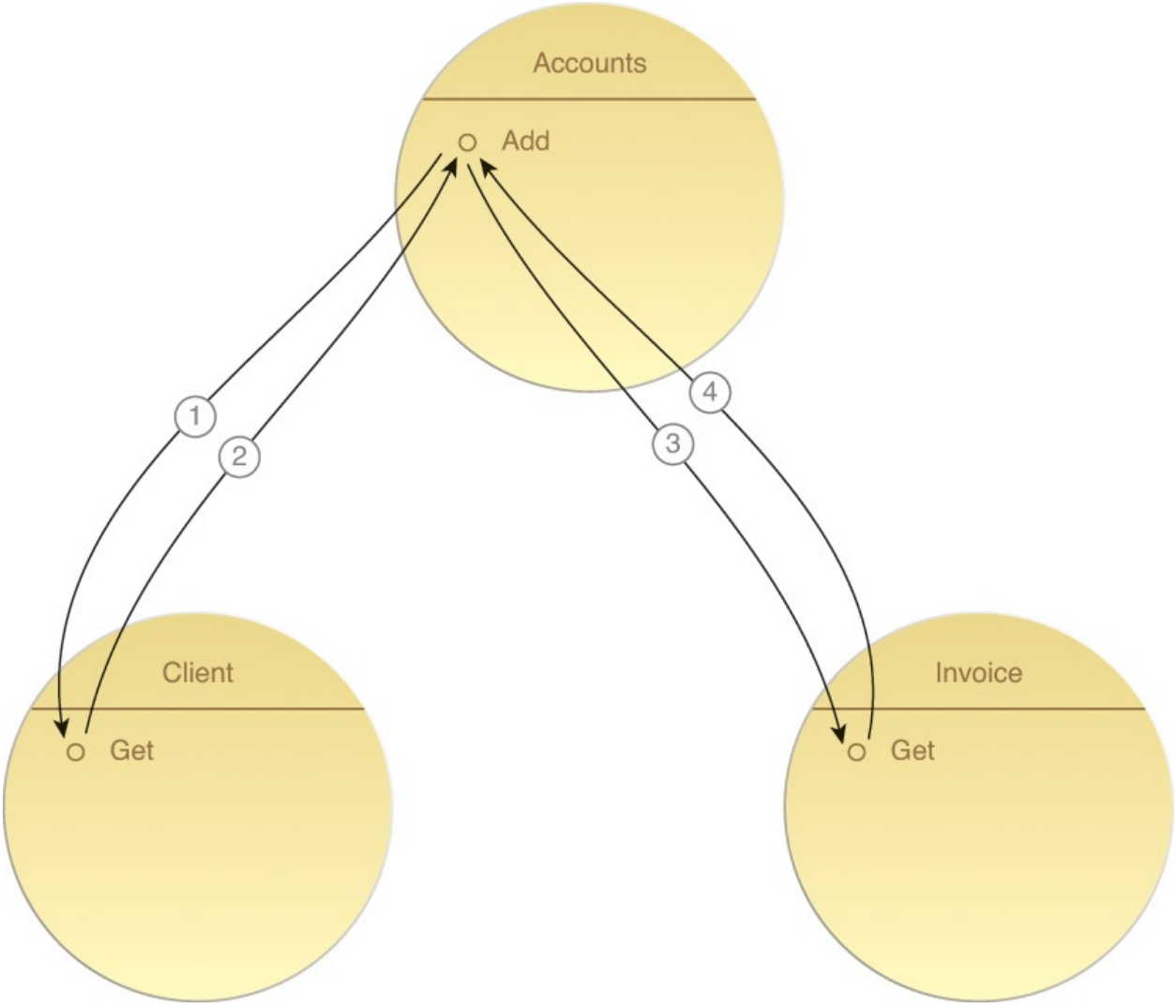


Figure 4.14 The Accounts service composition from a modeling perspective.

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

Each service composition has a corresponding *service composition architecture*. In much the same way an application architecture for a distributed system includes the individual architecture definitions of its components, this form of architecture encompasses the service architectures of all participating services (Figure 4.15).

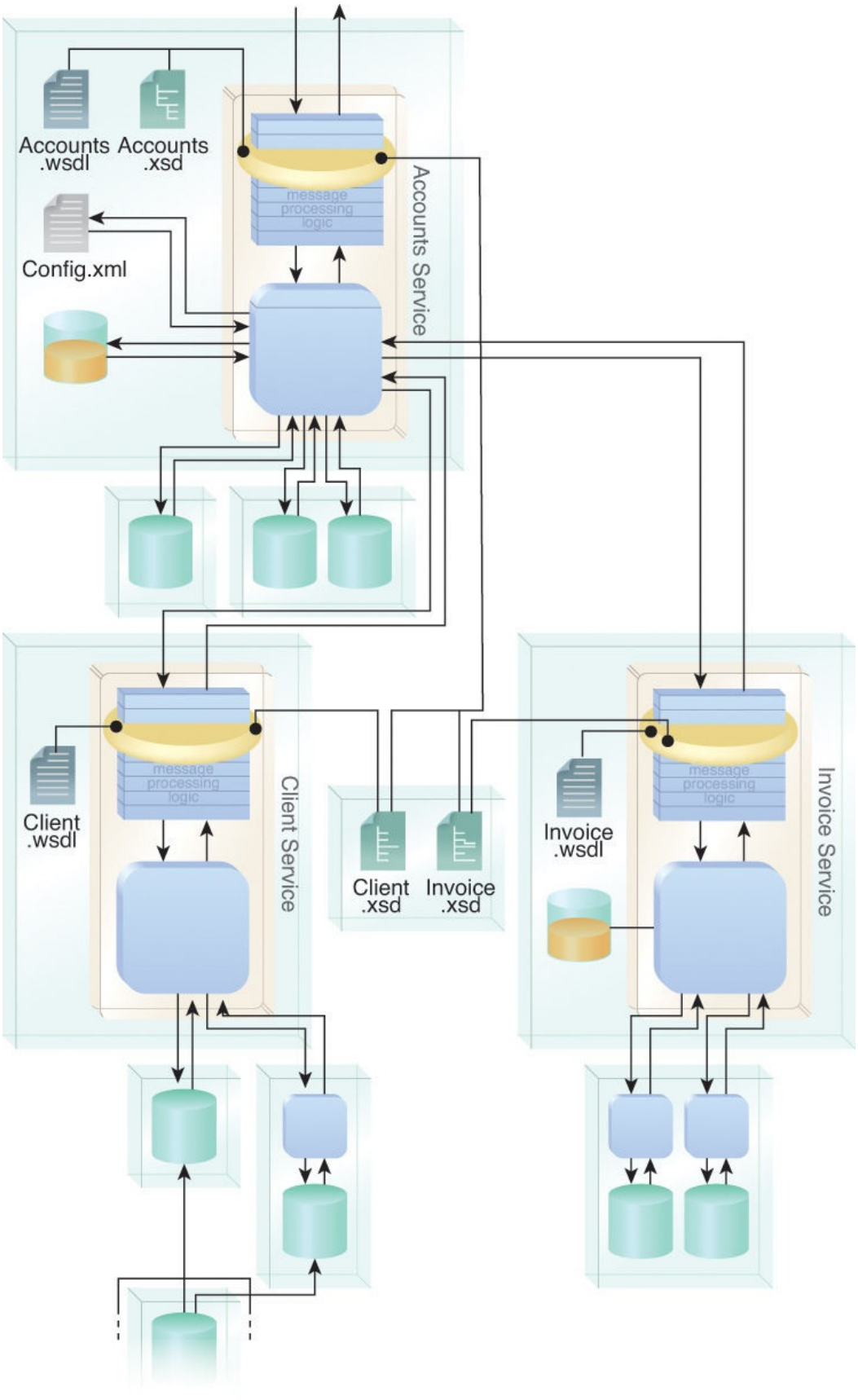

Accounts
.wsdl
Accounts
.xsd
Config.xml
message
processing
logic
Accounts Service
Client
.wsdl
message
processing
logic
Client Service
Client
.xsd
Invoice
.xsd
Invoice
.wsdl
message
processing
logic
Invoice Service

Figure 4.15 The same Accounts service composition from Figure 4.14 viewed from a physical architecture perspective illustrating how each composition member's underlying resources provide the functionality required to automate the process logic represented by the Accounts service's Add capability.

Note

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

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

For example, one difference in how composition architectures are documented is in the extent of detail they include about agnostic services involved in the composition. Because these types of service architecture specifications are often guarded—as per the requirements raised by the Service Abstraction (294) principle—a composition architecture may only be able to make reference to the technical interface documents and service-level agreement (SLA)-related information published as part of the service's public contract (Figure 4.16).

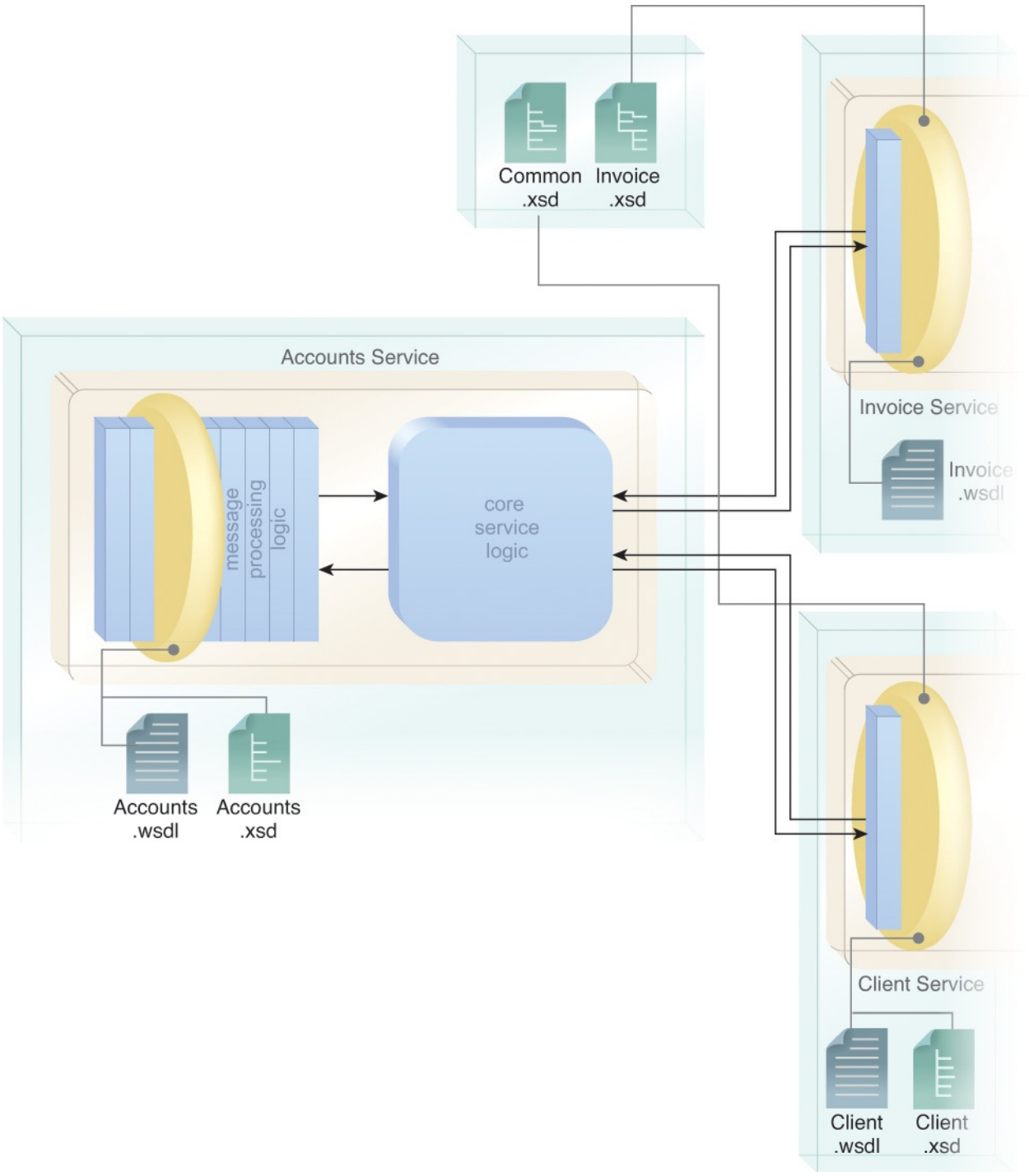


Figure 4.16 The physical service architecture view from Figure 4.15 is not available to the designer of the Accounts service. Instead, only the information published in the contracts for the Invoice and Client services can be accessed.

Another rather unique aspect of service composition architecture is that a composition may find itself a nested part of a larger parent composition, and

therefore one composition architecture may encompass or reference another (Figure 4.17).

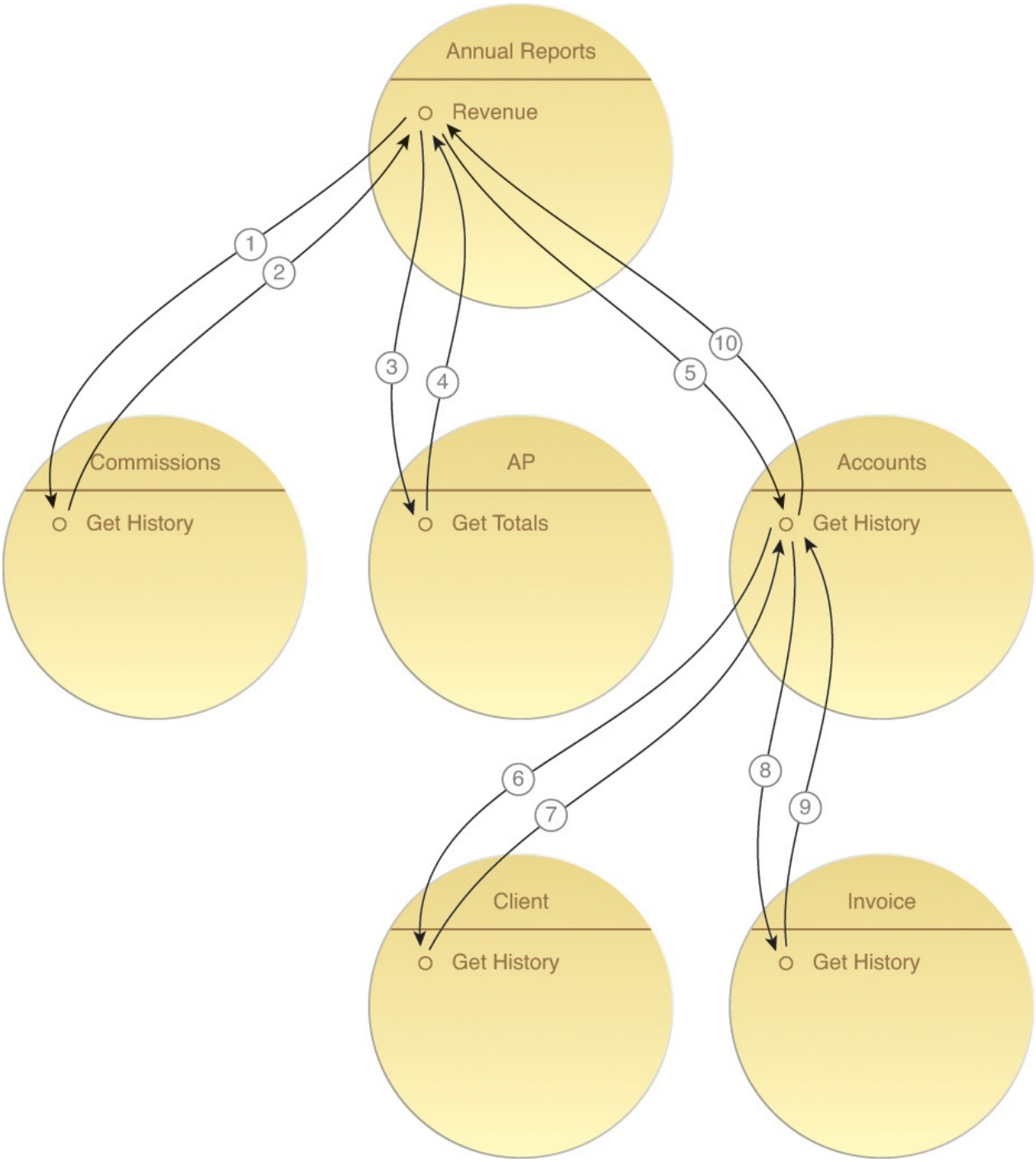


Figure 4.17 The Accounts service finds itself nested within the larger Annual Reports composition that composes the Accounts Get History capability which, in turn, composes capabilities within the Client and Invoice services.

Service composition architectures are much more than just an accumulation of individual service architectures (or contracts). A newly created composition is usually accompanied by a non-agnostic task service that is positioned as the

composition controller. The details of this service are less private, and its design is an integral part of the architecture because it provides the composition logic to invoke and interact with all identified composition members.

Furthermore, the business process the service is required to automate may involve the need for composition logic capable of dealing with multiple runtime scenarios (exception-related or otherwise), each of which may result in a different composition configuration. These scenarios and their related service activities and message paths are a common part of composition designs. They need to be understood and mapped out in advance so that the composition logic is fully prepared to deal with the range of runtime situations it may need to face (Figure 4.18).

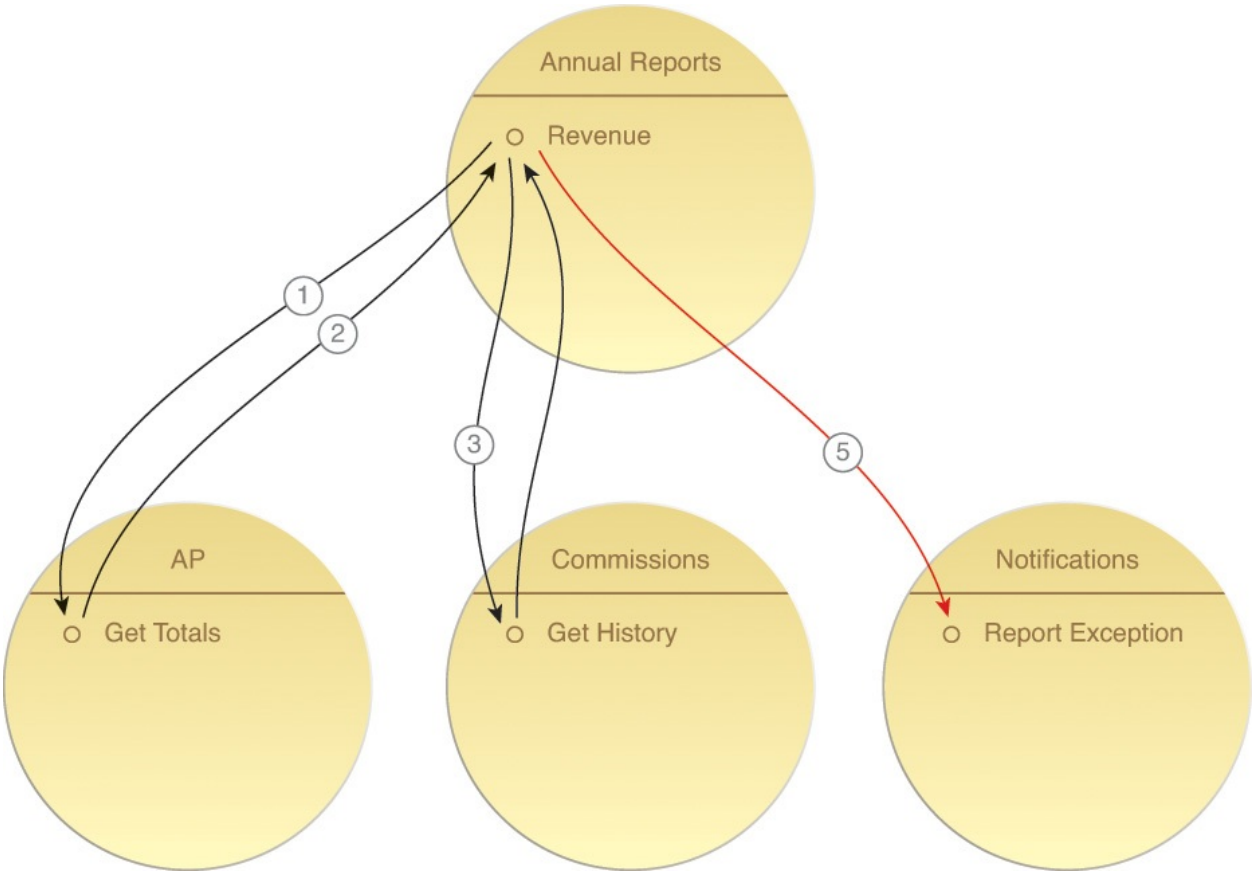


Figure 4.18 A given business process may need to be automated by a range of service compositions in order to accommodate different runtime scenarios. In this case, alternative composition logic within the Annual Report's Revenue capability kicks in to deal with an exception condition. As a result, the Notifications service is invoked prior to the Accounts service even being included in the composition.

Finally, the composition will rely on the activity management abilities of the

underlying runtime environment responsible for hosting the composition members. Security, transaction management, reliable messaging, and other infrastructure extensions, such as support for sophisticated message routing, may all find their way into a composition architecture specification.

SOA Patterns

Even though compositions are comprised of services, it is actually the service capabilities that are individually invoked and that execute a specific subset of service functionality to carry out the composition logic. This is why design patterns, such as Capability Composition [328] and Capability Recomposition [329] make specific reference to the composed capability (as opposed to the composed service).

Service Inventory Architecture

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

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

As explained in Chapter 3, a service inventory is a collection of independently standardized and governed services delivered within a pre-defined architectural boundary. This collection represents a meaningful scope that exceeds the processing boundary of a single business process and ideally spans numerous business processes.

SOA Patterns

The scope and boundary of a service inventory architecture can vary, as per the Enterprise Inventory [340] and Domain Inventory [338] patterns.

Ideally, the service inventory is first conceptually modeled, leading to the

creation of a service inventory blueprint. It is often this blueprint that ends up defining the required scope of the architecture type referred to as a service inventory architecture (Figure 4.19).

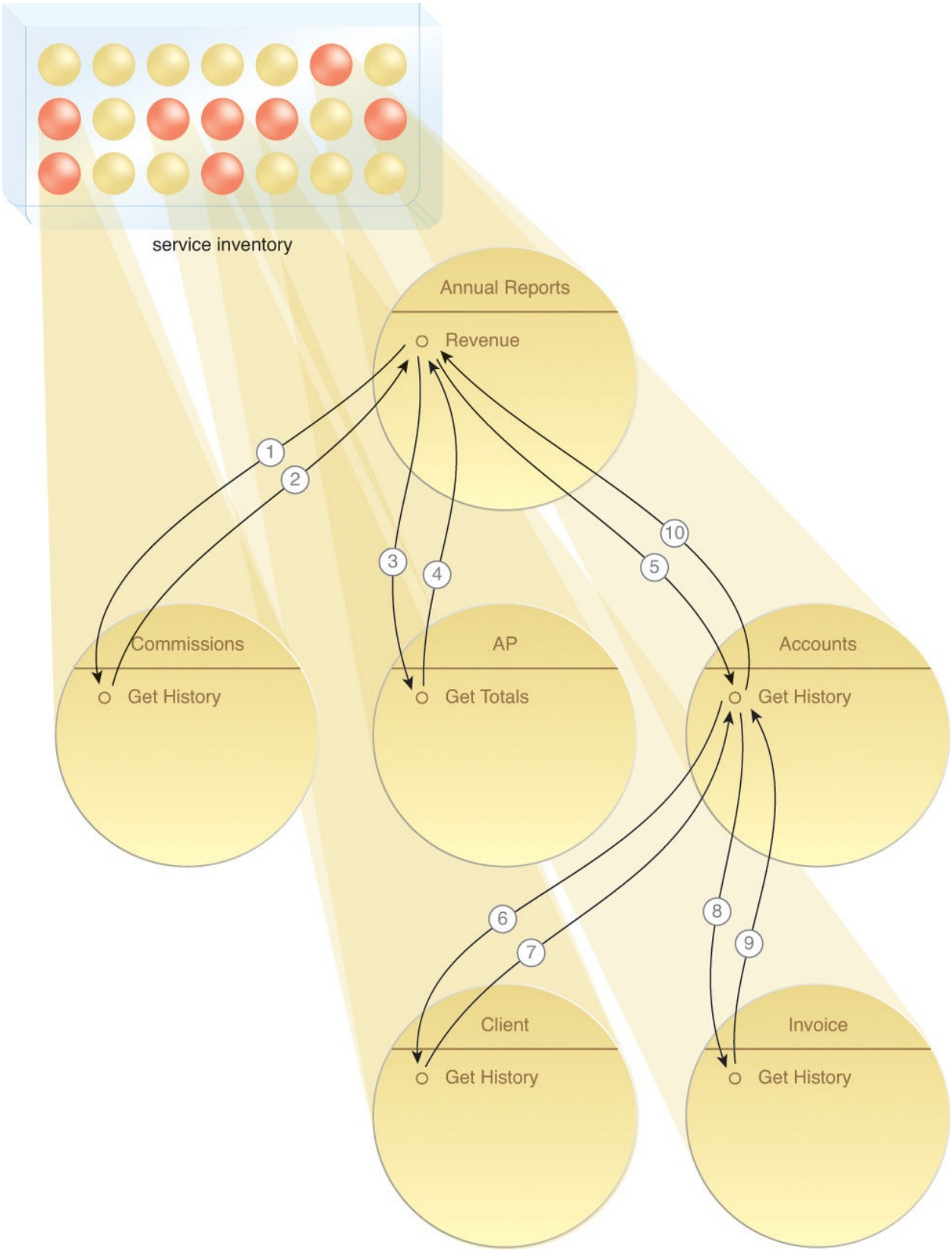


Figure 4.19 Ultimately, the services within an inventory can be composed and

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

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

As previously mentioned, the scope of a service inventory can be enterprise-wide, or it can represent a domain within the enterprise. For that reason, this type of architecture is not called a "domain architecture." It relates to the scope of the inventory boundary, which may encompass multiple domains.

SOA Patterns

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

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

Service-Oriented Enterprise Architecture

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

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

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

SOA Patterns

The Inventory Endpoint [346] pattern can play a key role when designing service inventory environments with external communication requirements.

Additionally, the service-oriented enterprise architecture can further establish enterprise-wide design standards and conventions to which all service, composition, and inventory architecture implementations need to comply, and which may also need to be referenced in the corresponding architecture specifications.

Note

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

Furthermore, additional types of service-oriented architecture can exist, especially when spanning beyond a private enterprise environment. Examples can include interbusiness service architecture, service-oriented community architecture and various hybrid architectures that encompass IT resources from external cloud computing environments.

4.3 The End Result of Service-Orientation and SOA

Business communities and the IT industry have an endless bi-directional relationship where each influences the other (Figure 4.20). Business demands and trends create automation requirements that the IT community strives to fulfill. New method and technology innovations produced by the IT community help inspire organizations to improve their existing business and even try out new lines of business. (The advent of cloud computing is a good example of the latter.)

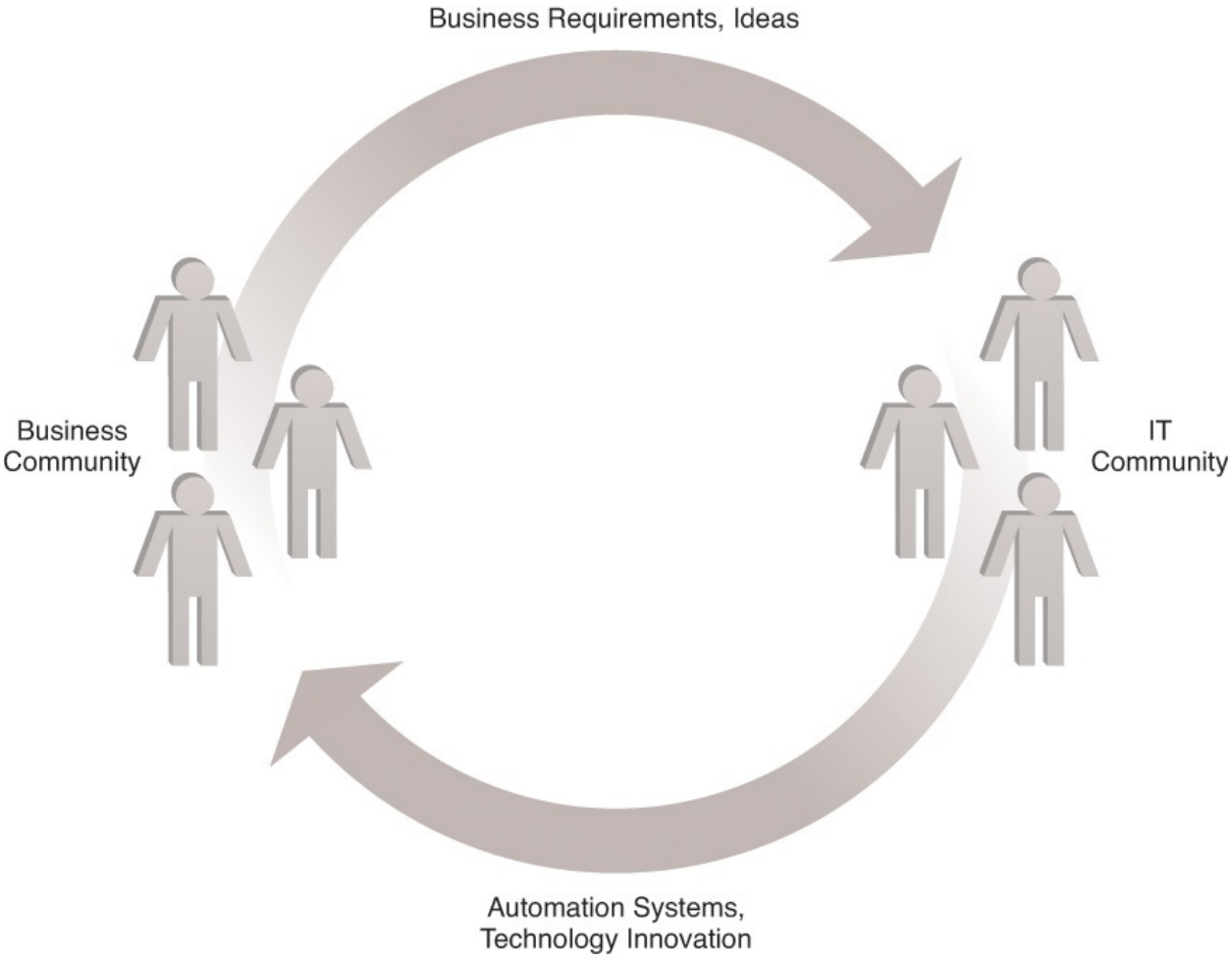


Figure 4.20 The endless progress cycle establishes the dynamics between the business and IT communities.

The IT industry has been through the cycle depicted in Figure 4.20 many times. Each iteration has brought about change and generally an increase in the sophistication and complexity of technology platforms.

Sometimes a series of iterations through this progress cycle leads to a foundational shift in the overall approach to automation and computing itself. The emergence of major platforms and frameworks, such as object-orientation and enterprise application integration, are examples of this. Significant changes like these represent an accumulation of technologies and methods and can therefore be considered landmarks in the evolution of IT itself. Each also results in the formation of distinct technology architecture requirements.

Service-oriented computing is no exception. The platform it establishes provides the potential to achieve significant strategic benefits that are a reflection of what business communities are currently demanding, as represented by the strategic goals and benefits previously described in Chapter 3.

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

How then does this relate to service-oriented technology architecture? Figure 4.21 hints at how the pursuit of these specific goals results in a series of impacts onto all architecture types brought upon by the application of service-orientation.

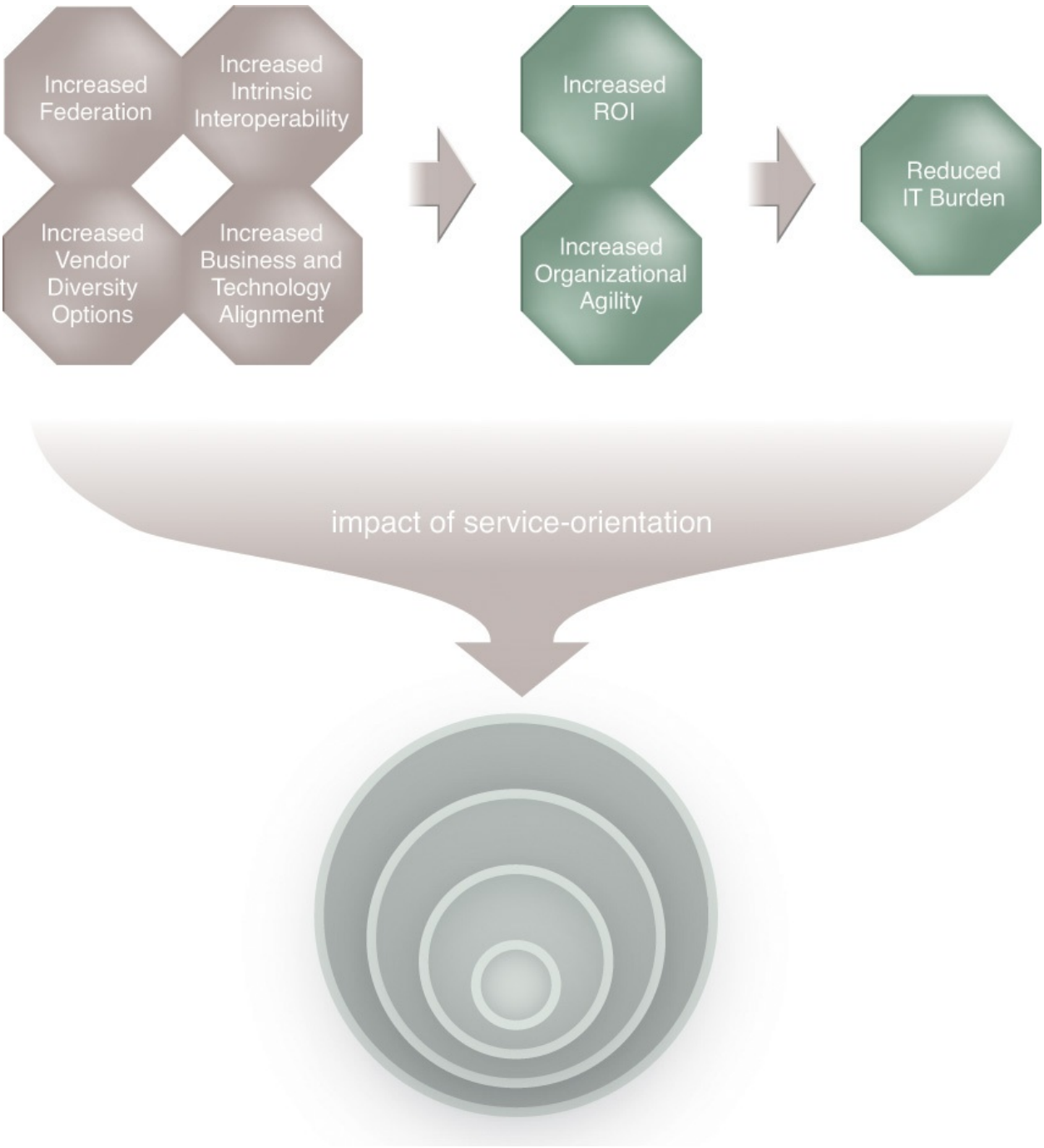


Figure 4.21 The common strategic goals and benefits of service-oriented computing are realized through the application of service-orientation. This, in turn, impacts the demands and requirements placed upon the four types of service-oriented technology architectures. (Note that the three goals on the right represent the ultimate target benefits sought in a typical SOA initiative.)

Note

> For those of you interested in how each of the strategic goals specifically influences the four types of service-oriented architecture, Chapter 23 in *SOA Design Patterns* documents the individual impacts.

Ultimately, the successful implementation of service-oriented architectures will support and maintain the benefits associated with the strategic goals of service-oriented computing. As illustrated in Figure 4.22, the progress cycle that continually transpires between business and IT communities results in constant change. Standardized, optimized, and overall robust service-oriented architectures fully support and even enable the accommodation of this change as a natural characteristic of a service-oriented enterprise.

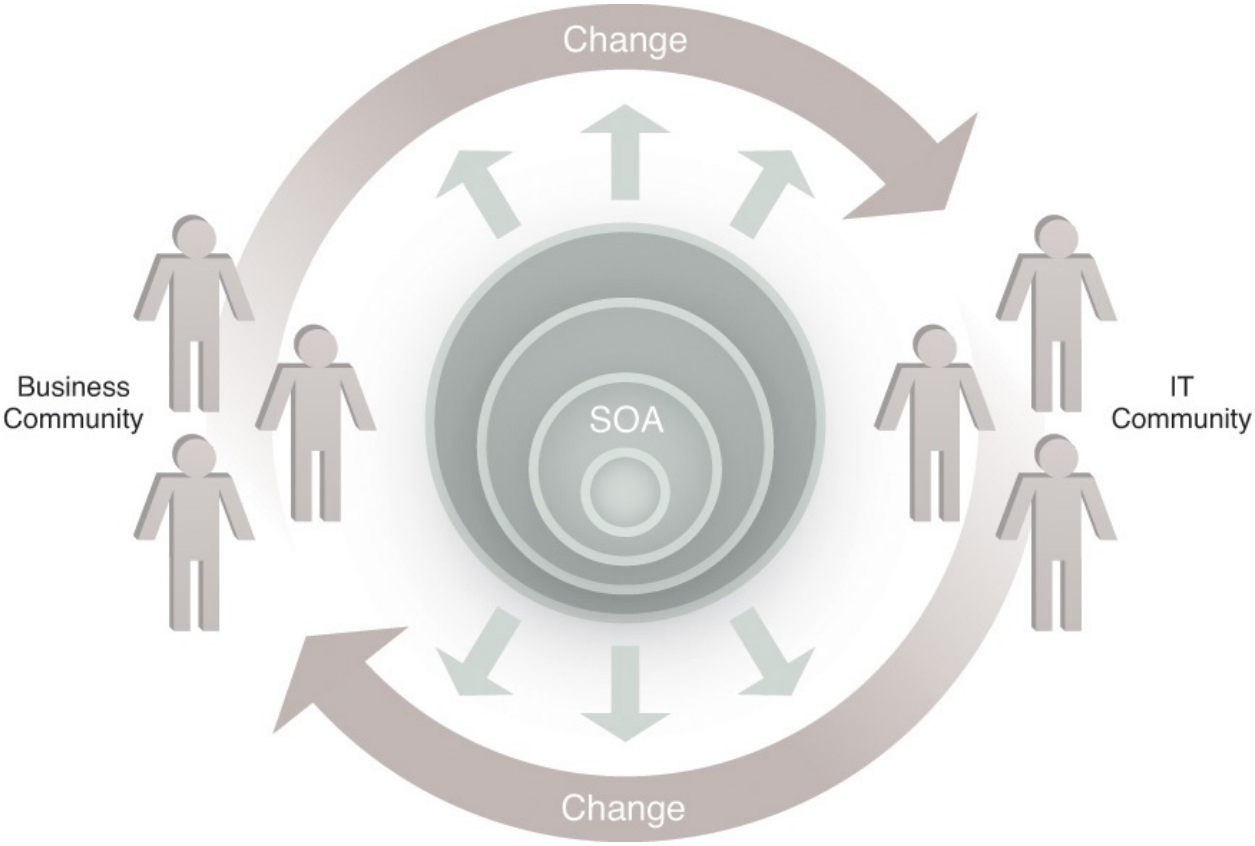


Figure 4.22 Service-oriented technology architecture supports the two-way dynamic between business and IT communities, allowing each to introduce or accommodate change throughout an endless cycle.

Finally, to best understand how to achieve a technology architecture capable of enabling the two-way dynamic illustrated in Figure 4.22, we need to reveal how, behind the scenes, the supporting, formalized bodies of knowledge and intelligence comprise SOA as a mature field of practice (Figure 4.23).

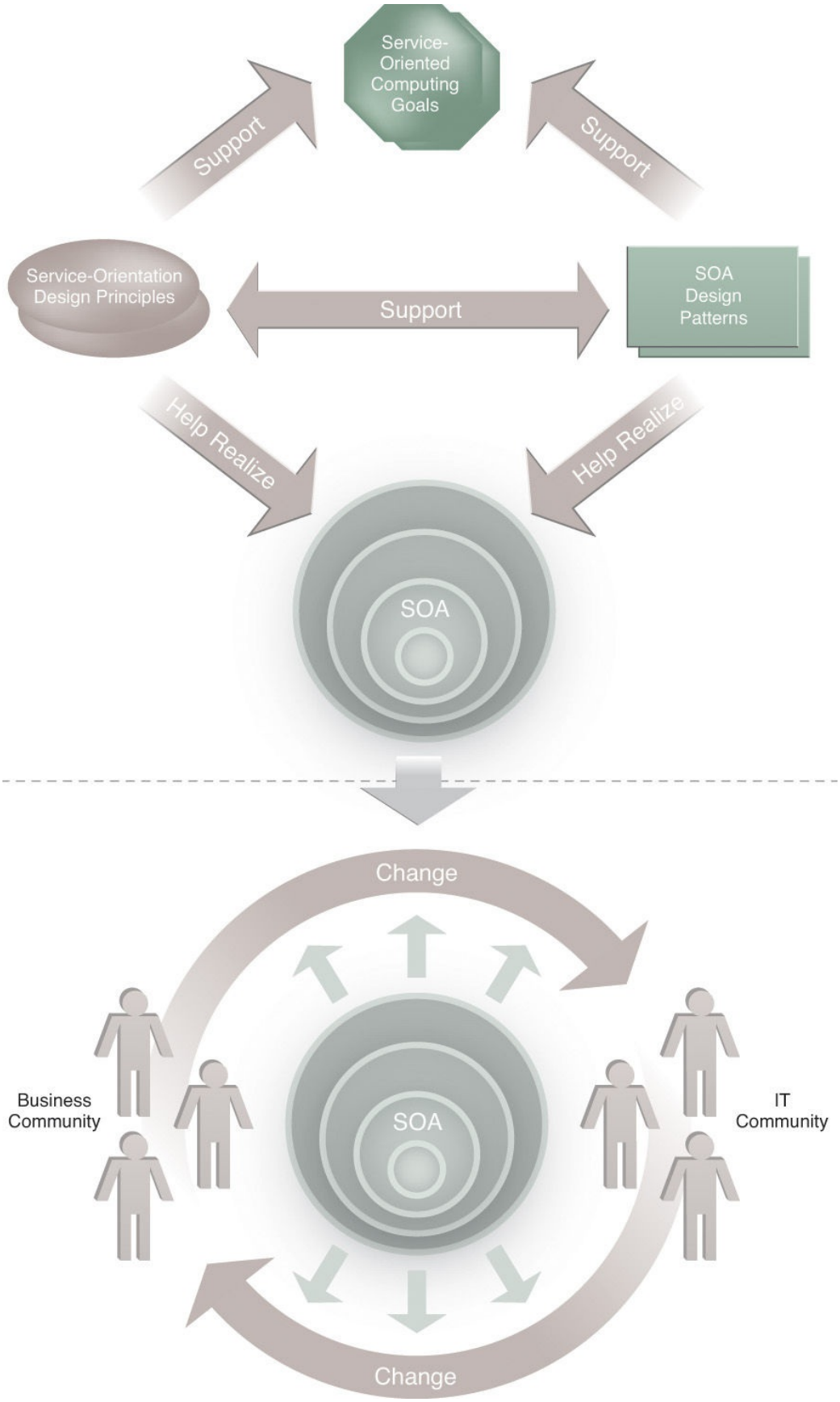

Service-Oriented Computing Goals
Support
Support
Service-Orientation Design Principles
Support
SOA Design Patterns
Help Realize
Help Realize
SOA
Change
Business Community
SOA
IT Community
Change

Figure 4.23 The strategic goals of service-oriented computing represent a target state that can be achieved through a method provided by service-orientation. The successful application of service-orientation principles and supporting SOA design patterns helps to shape and define requirements for different types of service-oriented architectures, resulting in an IT automation model that is designed to fully support the two-way cycle of change through which business and IT communities continually transition.

4.4 SOA Project and Lifecycle Stages

Understanding how to realize service-oriented architecture also requires an understanding of how SOA projects are carried out. For the remainder of this chapter, we take a step away from technology to briefly summarize common SOA methodology and project delivery topics.

Note

This section provides a good transition to Chapter 5, which explores service definition as a foundational part of the service-oriented analysis project stage, and Chapters 6 to 9, which further delve into the service-oriented analysis stage and then cover considerations pertaining to the service-oriented design project stage.

Methodology and Project Delivery Strategies

Several project delivery approaches can be employed to build services. The bottom-up strategy, for example, is tactically focused in that it makes the fulfillment of immediate business requirements a priority and the prime objective of the project. On the other side of the spectrum is the top-down strategy, which advocates the completion of an inventory analysis prior to the actual design, development, and delivery of services.

As shown in Figure 4.24, each approach has its own benefits and consequences. Whereas the bottom-up strategy avoids the extra cost, effort, and time required to deliver services via a top-down approach, it ends up imposing increased governance burden because bottom-up delivered services tend to have shorter lifespans and require more frequent maintenance and refactoring.

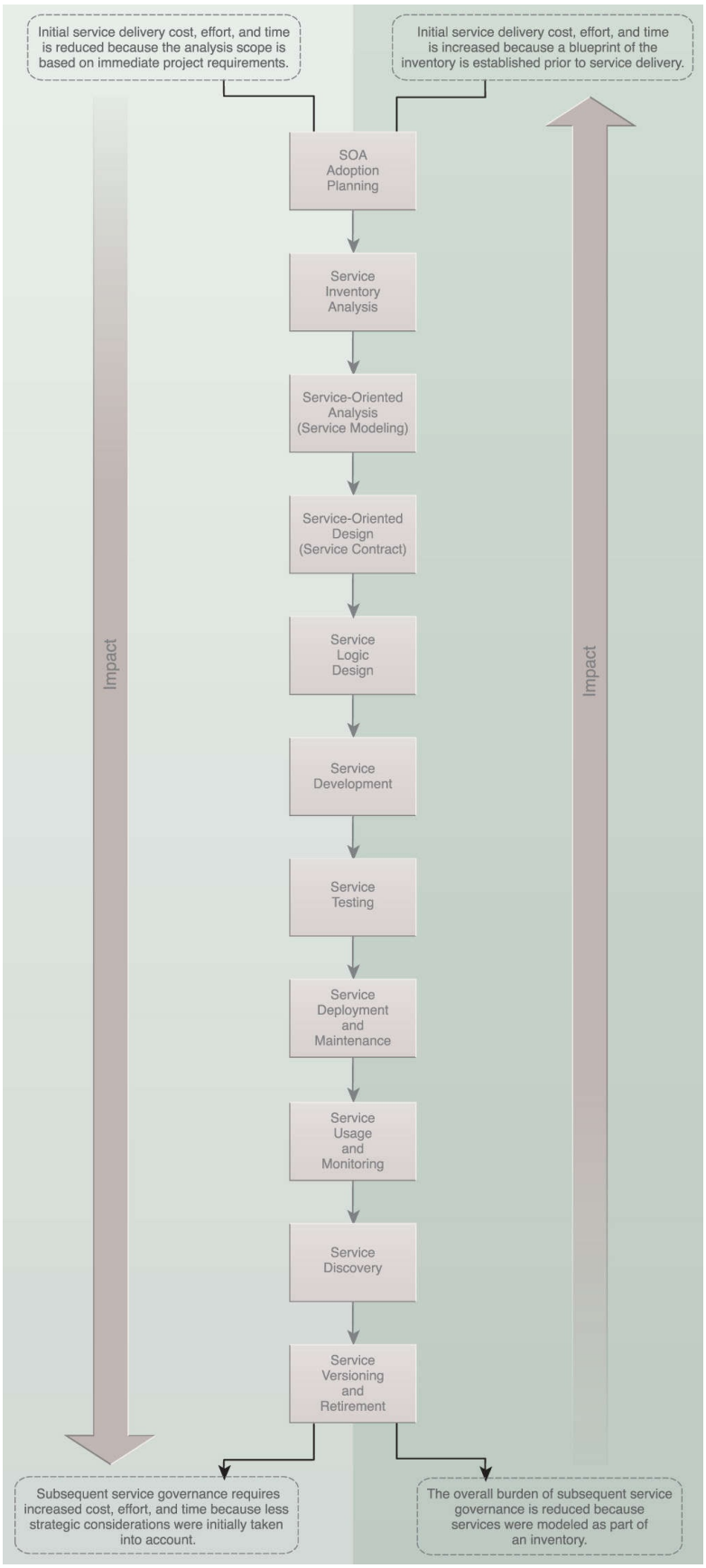

Initial service delivery cost, effort, and time is reduced because the analysis scope is based on immediate project requirements.
Initial service delivery cost, effort, and time is increased because a blueprint of the inventory is established prior to service delivery.
SOA Adoption Planning
Service Inventory Analysis
Service-Oriented Analysis (Service Modeling)
Service-Oriented Design (Service Contract)
Service Logic Design
Service Development
Service Testing
Service Deployment and Maintenance
Service Usage and Monitoring
Service Discovery
Service Versioning and Retirement
Impact
Impact
Subsequent service governance requires increased cost, effort, and time because less strategic considerations were initially taken into account.
The overall burden of subsequent service governance is reduced because services were modeled as part of an inventory.

Figure 4.24 Generally, the less time and effort spent on the upfront service analysis, the greater the ongoing, post-deployment governance burden. The approach on the left is comparable with bottom-up service delivery and the approach on the right is more akin to top-down delivery. SOA methodologies that attempt to combine elements of both approaches also exist.

The top-down strategy demands more of an initial investment because it introduces an upfront analysis stage focused on the creation of the service inventory blueprint. A collection of service candidates are individually defined as part of this blueprint to ensure that subsequent service designs will be highly normalized, standardized, and aligned.

Note

A top-down strategy needs to be applied to an extent to meaningfully carry out the service-oriented analysis and service-oriented design stages covered in Chapters 6 to 9. The scope of this effort is determined by the scope of the planned service inventory, as per the Balanced Scope pillar covered in Chapter 3.

SOA Project Stages

Figure 4.25 displays the common and primary stages related to SOA project delivery and the overall service delivery lifecycle. Although the stages are shown sequentially, how and when each stage is carried out depends on the methodology being used. Different methodologies can be considered, depending on the nature and scope of the overall SOA project, the size and extent of standardization of the service inventory for which services are being delivered, and the manner in which tactical (short-term) requirements are being prioritized in relation to strategic (long-term) requirements.

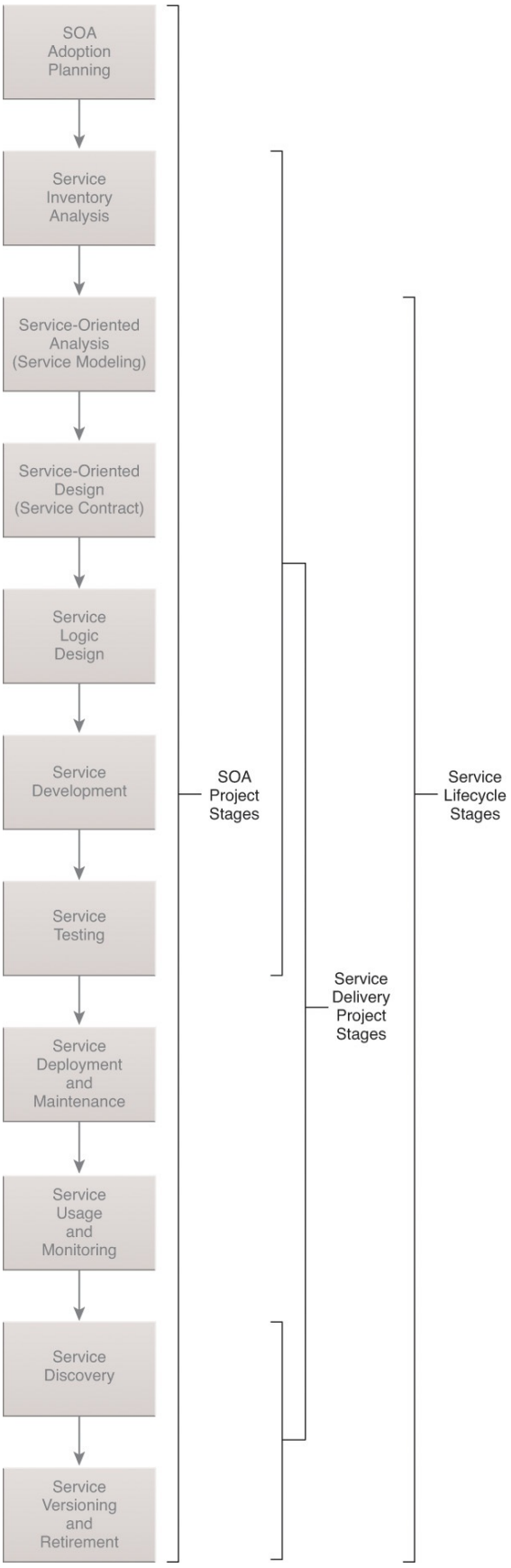

SOA Adoption Planning
Service Inventory Analysis
Service-Oriented Analysis (Service Modeling)
Service-Oriented Design (Service Contract)
Service Logic Design
Service Development
Service Testing
Service Deployment and Maintenance
Service Usage and Monitoring
Service Discovery
Service Versioning and Retirement
SOA Project Stages
Service Delivery Project Stages
Service Lifecycle Stages

Figure 4.25 Common stages associated with SOA projects. Note the distinction between SOA project stages, service delivery project stages, and service lifecycle stages. These terms are used in subsequent chapters when referring to the overall adoption project, the delivery of individual services, and service-specific lifecycle stages, respectively.

Top-down SOA projects tend to emphasize the need for some meaningful extent of the strategic target state that the delivery of each service is intended to support. In order to realize this, some level of increased upfront analysis effort is generally necessary. Therefore, a primary way in which SOA project delivery methodologies differ is in how they position and prioritize analysis-related phases.

There are two primary analysis phases in a typical SOA project: the analysis of individual services in relation to business process automation, and the collective analysis of a service inventory. The service-oriented analysis phase is dedicated to producing conceptual service definitions (service candidates) as part of the functional decomposition of business process logic. The service inventory analysis establishes a cycle whereby the service-oriented analysis process is carried out iteratively (together with other business processes) to whatever extent a top-down (strategic) approach is followed.

The upcoming sections briefly describe these and other stages.

SOA Adoption Planning

During this initial stage is when foundational planning decisions are made. These decisions will shape the entire project, which is why this is considered a critical stage that may require separately allocated funding and time to carry out significant studies required to assess and determine a range of factors, including:

- Scope of planned service inventory and the ultimate target state
- Milestones representing intermediate target states
- Timeline for the completion of milestones and the overall adoption effort
- Available funding and suitable funding model
- Governance system
- Management system
- Methodology
- Risk assessment

Additionally, prerequisite requirements need to be defined in order to establish

criteria used to determine the overall viability of the SOA adoption. The basis of these requirements typically originates with the four pillars of service-orientation described earlier in Chapter 3.

Service Inventory Analysis

The scope of a service inventory is expected to be meaningfully "cross-silo," which generally implies that it encompasses multiple business processes or operational areas within an organization.

This service inventory analysis stage is dedicated to conceptually defining an inventory of services. It is comprised of a cycle (Figure 4.26) during which the service-oriented analysis stage (explained shortly) is carried out once during each iteration. Each completion of a service-oriented analysis results in the definition of new service candidates or the refinement of existing ones. The cycle is repeated until all business processes that fall within the domain of the service inventory are analyzed and decomposed into individual actions suitable for service encapsulation.

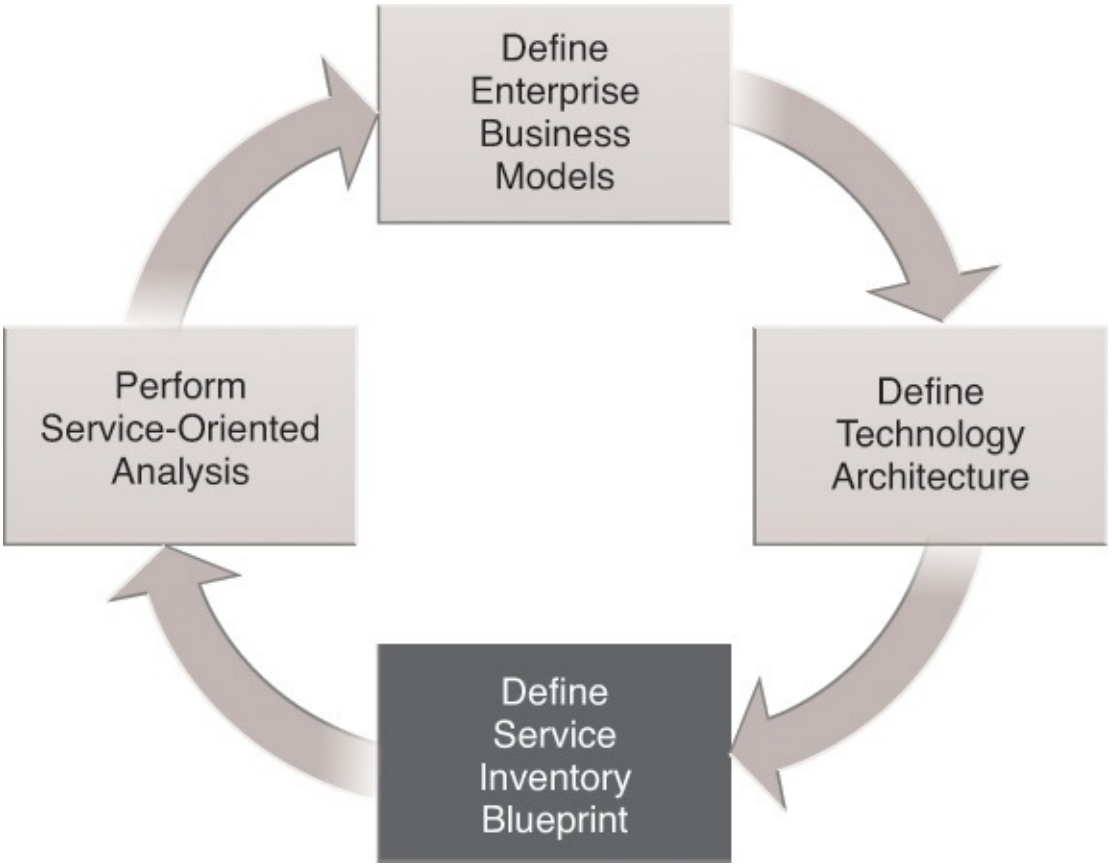


Figure 4.26 The service inventory analysis cycle. The highlighted step refers to the service inventory blueprint that represents the primary deliverable of

this stage.

As individual service candidates are identified, they are assigned appropriate functional contexts in relation to each other. This ensures that services (within the service inventory boundary) are normalized so that they don't functionally overlap. As a result, service reuse is maximized and the separation of concerns is cleanly carried out. A primary deliverable produced during this stage is the *service inventory blueprint*.

The scope of the initiative and the size of the target service inventory tend to determine the amount of upfront effort required to create a complete service inventory blueprint. More upfront analysis results in a better defined conceptual blueprint, which is intended to lead to the creation of a better quality inventory of services. Less upfront analysis leads to partial or less well-defined service inventory blueprints.

Here are brief descriptions of the primary analysis cycle steps:

- *Define Enterprise Business Models* – Business models and specifications (such as business process definitions, business entity models, logical data models, etc.) are identified, defined, and, if necessary, brought up-to-date and further refined. These models are used as the primary business analysis input.
- *Define Technology Architecture* – Based on what we learn of business automation and service encapsulation requirements, we are able to define preliminary technology architecture characteristics and constraints. This provides a preview of the service inventory environment, which can raise practical considerations that may impact how we define service candidates.
- *Define Service Inventory Blueprint* – After an initial definition that establishes the scope and structure of the planned service inventory, this blueprint acts as the master specification wherein modeled service candidates are documented.
- *Perform Service-Oriented Analysis* – Each iteration of the service inventory lifecycle executes a service-oriented analysis process.

The service inventory blueprint is incrementally defined as a result of repeated iterations of steps that include the service-oriented analysis.

Note

The scope of the service inventory analysis stage and the resulting service inventory blueprint directly relates to the Balanced Scope

consideration explained in the *The Four Pillars of Service-Orientation* section in Chapter 3, as well as the possible application of the Domain Inventory [338] pattern.

Service-Oriented Analysis (Service Modeling)

A fundamental characteristic of SOA projects is that they emphasize the need for working toward a strategic target state that the delivery of each service is intended to support. To realize this, some level of increased upfront analysis effort is generally necessary. Therefore, a primary way in which SOA project delivery methodologies differ is in how they position and prioritize analysis-related phases.

Service-oriented analysis represents one of the early stages in an SOA initiative and the first phase in the service delivery cycle (Figure 4.27). It is a process that begins with preparatory information-gathering steps completed in support of a service modeling subprocess.

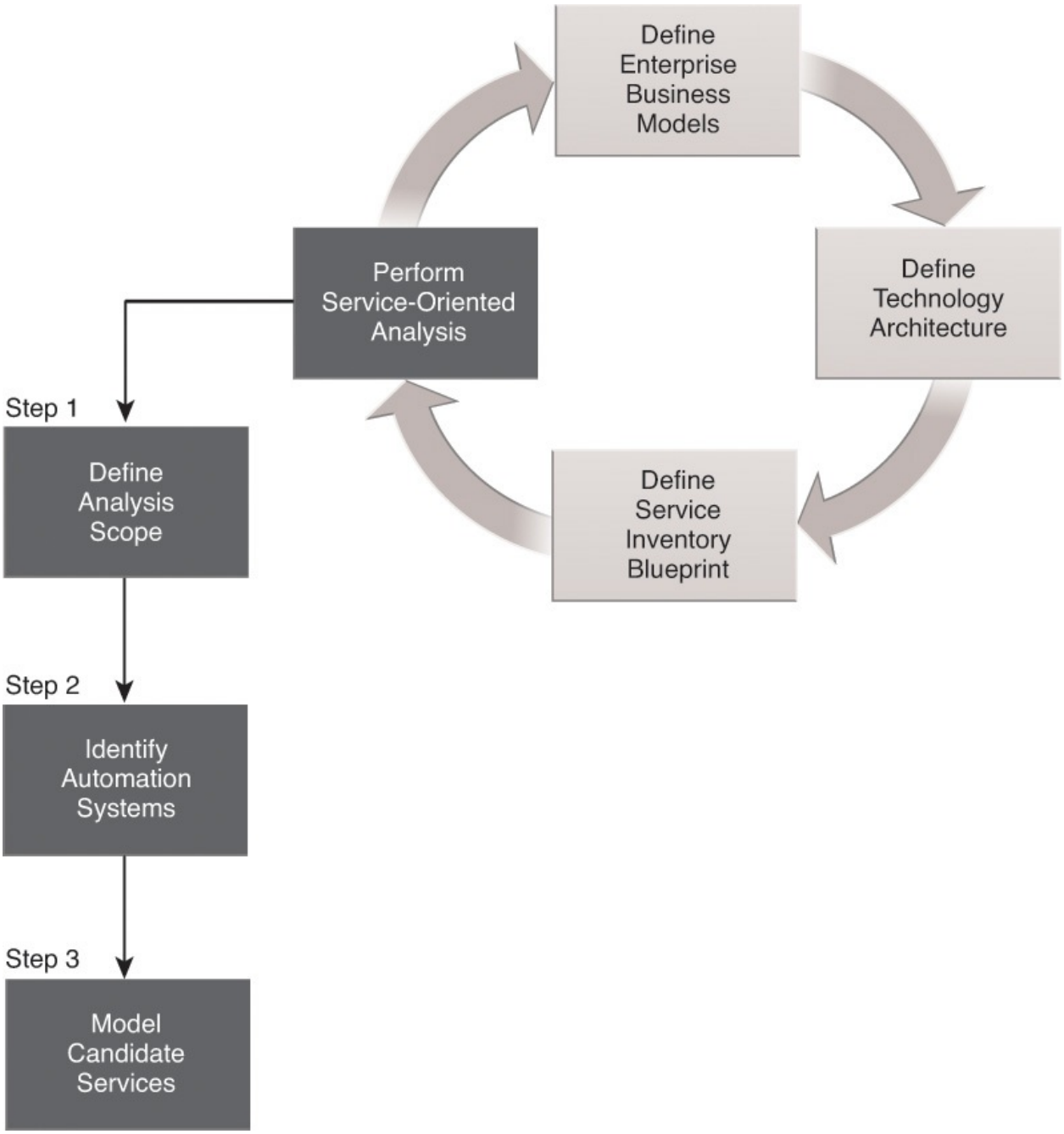


Figure 4.27 A generic service-oriented analysis process in which the first two steps collect information in preparation for a detailed service modeling subprocess represented by the Model Candidate Services step.

The service-oriented analysis process is generally carried out iteratively, once for each business process. Typically, the delivery of a service inventory determines a scope that represents a meaningful domain of the enterprise (as per the Balanced Scope pillar discussed in Chapter 3), or even the enterprise as a whole. All iterations of the service-oriented analysis then pertain to that scope, with each

iteration contributing to the service inventory blueprint.

Steps 1 and 2 essentially represent information-gathering tasks that are carried out in preparation for the modeling process performed in Step 3.

Step 1: Define Business Automation Requirements

Through whatever means business requirements are normally collected, their documentation is required for this analysis process to begin. Given that the scope of our analysis centers around the creation of services in support of a service-oriented solution, only requirements related to the scope of that solution should be considered.

Business requirements should be sufficiently mature so that a high-level automation process can be defined. This business process documentation will be used as the starting point of a service modeling process.

Step 2: Identify Existing Automation Systems

Existing utility logic that is already, to whatever extent, automating any of the requirements identified in Step 1 needs to be identified. Although a service-oriented analysis will not determine exactly how Web services will encapsulate or replace legacy utility logic, it does assist us in providing some scope of the systems potentially affected.

The details of how Web services or REST services relate to existing systems are ironed out in the service-oriented design phase. For now, this information will be used to help identify utility service candidates during the service modeling process.

Note that this step is tailored toward supporting the modeling efforts of larger-scaled service-oriented solutions. An understanding of affected legacy environments is still useful when modeling a smaller amount of services, which does not require substantial research efforts.

Step 3: Model Candidate Services

A service-oriented analysis introduces the concept of *service modeling*, a process by which service operation candidates are identified and then grouped into a logical context. These groups eventually take shape as service candidates that are then further assembled into a tentative composite model representing the combined logic of the planned service-oriented application.

Note

Chapters 6 and 7 provide service modeling processes for Web services and REST services, respectively.

A key success factor of the service-oriented analysis process is the hands-on collaboration of both business analysts and technology architects (Figure 4.28). The former group is especially involved in the definition of service candidates within a business-centric functional context because they understand the business processes used as input for the analysis and because service-orientation aims to align business and IT more closely.

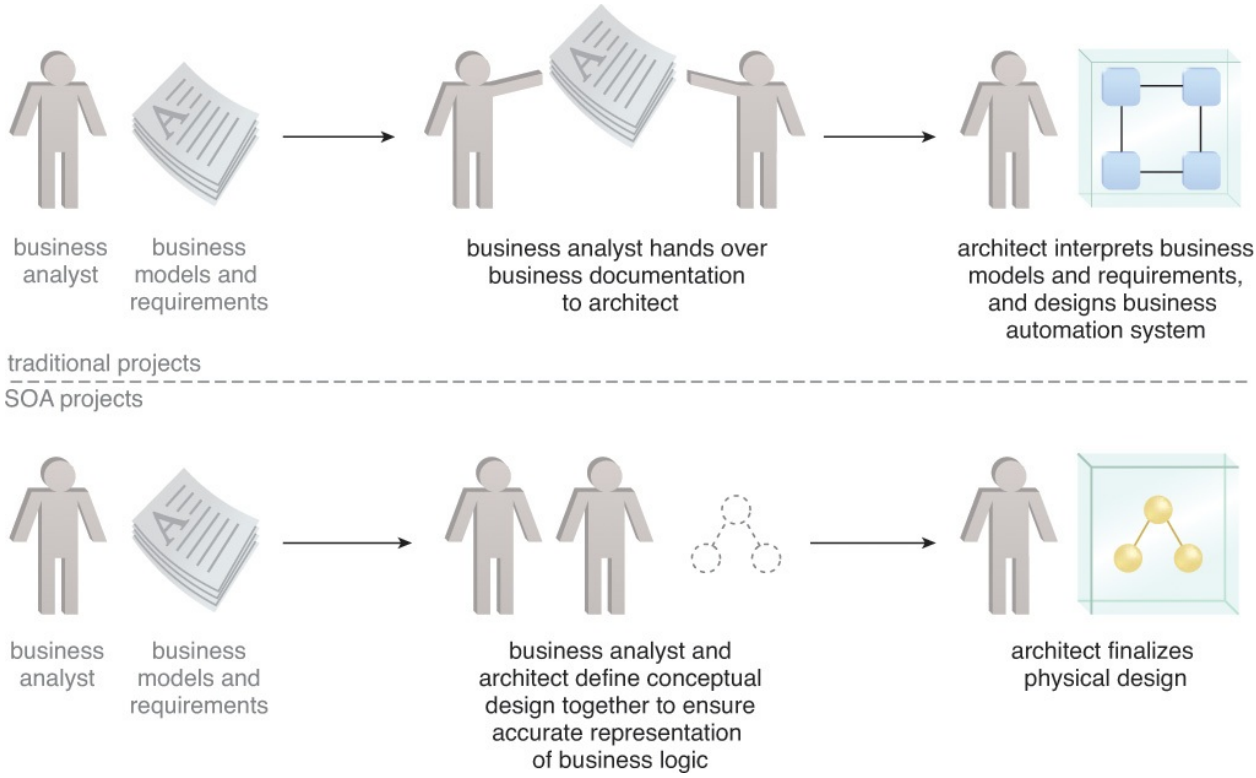


Figure 4.28 A look at how the collaboration between business analysts and technology architects changes with SOA projects. While the depicted collaborative relationship between business analysts and architects may not be unique to an SOA project, the nature and scope of the analysis process are.

Service-Oriented Design (Service Contract)

The service-oriented design phase represents a service delivery lifecycle stage dedicated to producing service contracts in support of the well-established "contract-first" approach to software development (Figure 4.29).

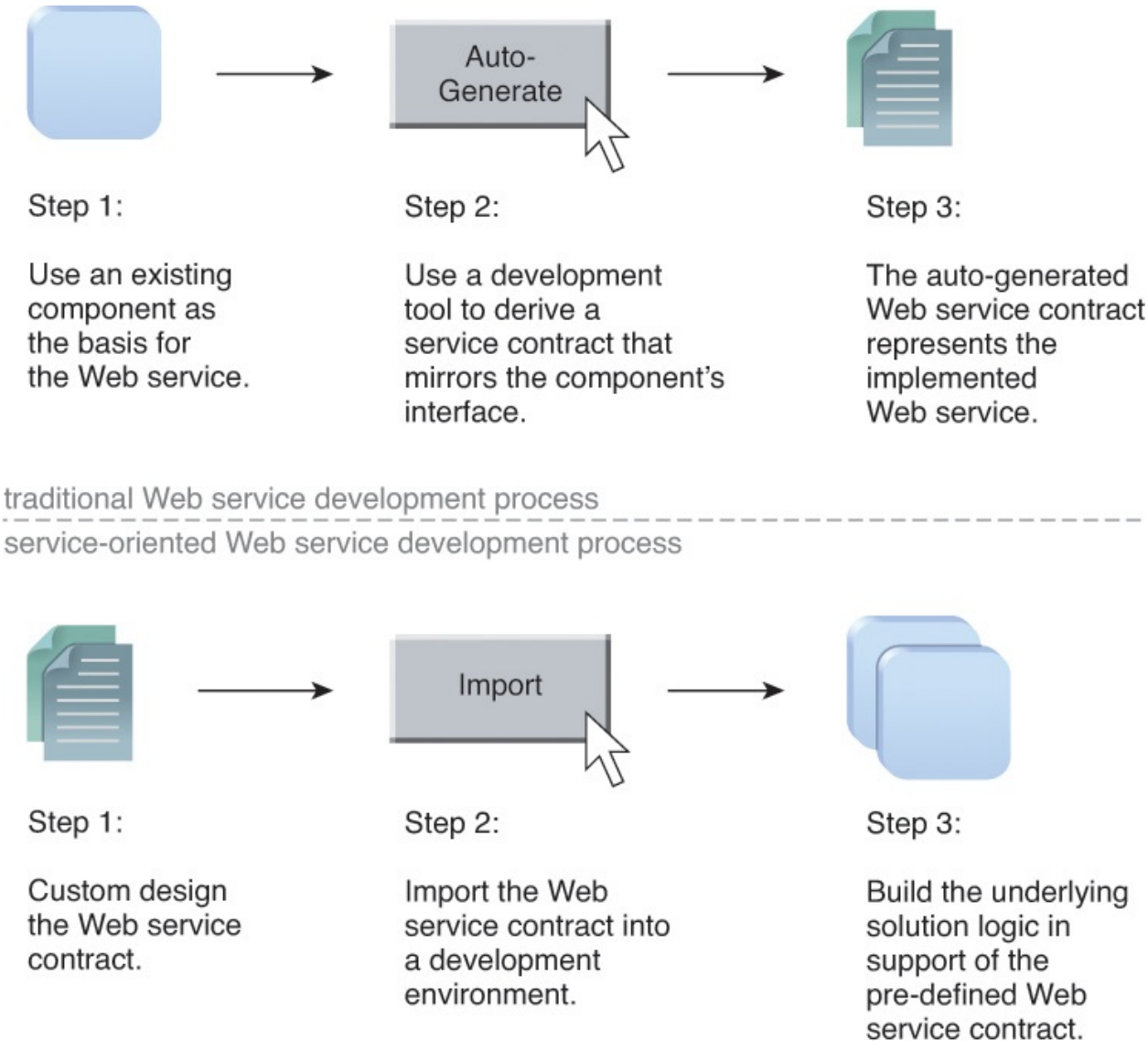


Figure 4.29 Unlike the popular process of deriving Web service contracts from existing components, SOA advocates a specific approach that encourages us to postpone development until after a custom designed, standardized contract is in place.

The typical starting point for the service-oriented design process is a service candidate that was produced as a result of completing all required iterations of the service-oriented analysis process (Figure 4.30). Service-oriented design subjects this service candidate to additional considerations that shape it into a technical service contract in alignment with other service contracts being produced for the same service inventory.

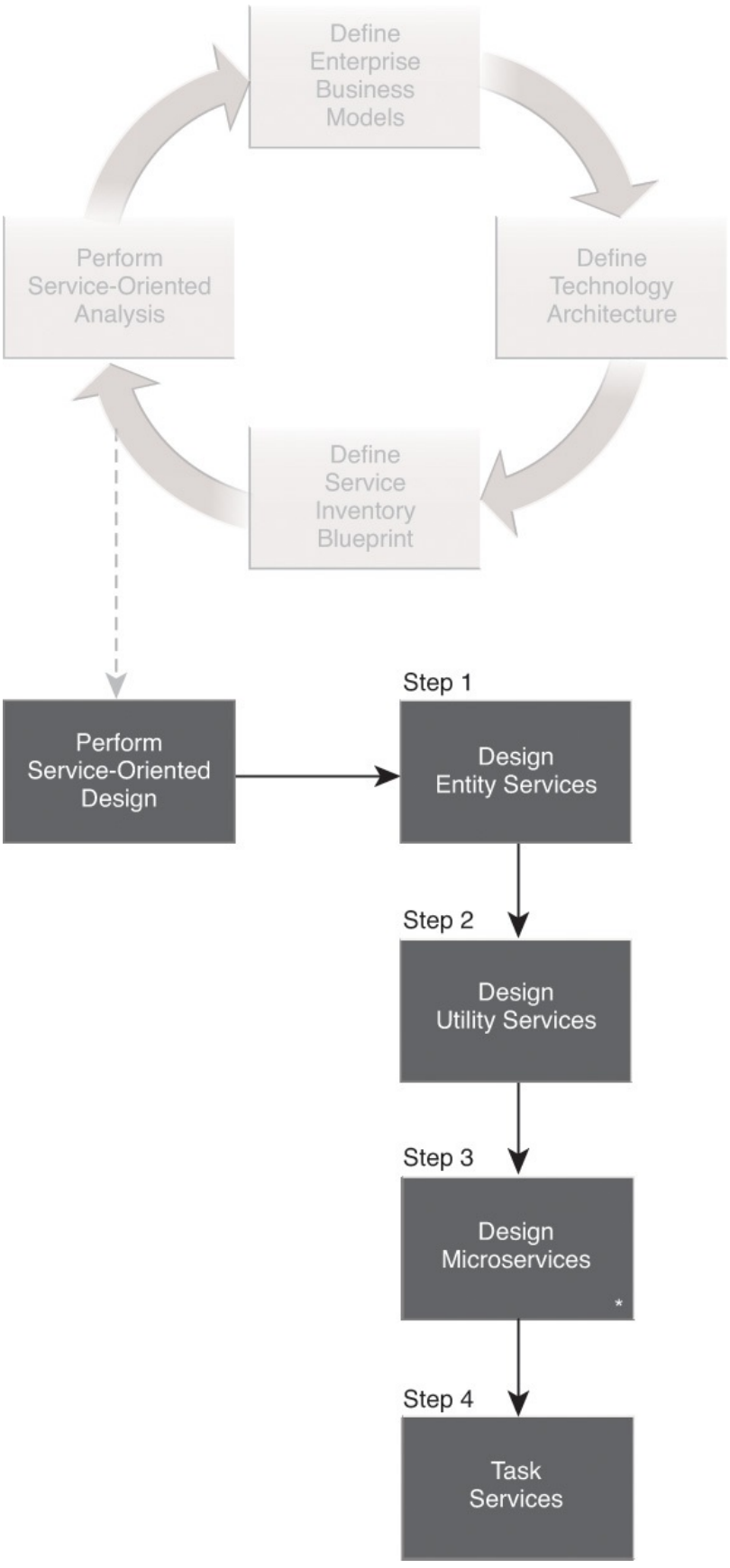

Define Enterprise Business Models
Define Technology Architecture
Define Service Inventory Blueprint
Perform Service-Oriented Analysis
Perform Service-Oriented Design
Step 1
Design Entity Services
Step 2
Design Utility Services
Step 3
Design Microservices
*
Step 4
Task Services

Figure 4.30 Subsequent to the analysis effort, services are subjected to a service-oriented design process.

As a precursor to the service logic design stage, service-oriented design is comprised of a process that ushers service architects through a series of considerations to ensure that the service contract being produced fulfills business requirements while representing a normalized functional context that further adheres to service-orientation principles. Part of this process further includes the authoring of the SLA, which may especially be of significance for cloud-based services being offered to a broader consumer base.

Service Logic Design

By preceding the design of service logic with the service-oriented design process, the service contract is established and finalized prior to the underlying service architecture and the logic that will be responsible for carrying out the functionality expressed in the service contract. This deliberate sequence of project stages is in support of the Standardized Service Contract (291) principle, which states that service contracts should be standardized in relation to each other within a given service inventory boundary.

How service logic is designed is dictated by the business automation requirements that need to be fulfilled by the service. With service-oriented solutions, a given service may be able to address business requirements individually or, more commonly, as part of a service composition.

Service Development

After all design specifications have been completed, the actual programming of the service can begin. Because the service architecture will already have been well-defined as a result of the previous stages and the involvement of custom design standards, service developers will generally have clear direction as to how to build the various parts of the service architecture.

Service Testing

Services need to undergo the same types of testing and quality assurance cycles as traditional custom-developed applications. However, new requirements introduce the need for additional testing methods and effort. For example, to support the realization of the Service Composability (302) principle, newly delivered services need to be tested individually and as part of service compositions. Agnostic services that provide reusable logic especially require

rigorous testing to ensure that they are ready for repeated usage (both concurrently as part of the same service compositions and by different service compositions).

The following are examples of common Service Testing considerations:

- What types of service consumers could potentially access a service?
- Will the service need to be deployed in a cloud environment?
- What types of exception conditions and security threats could a service be potentially subjected to?
- Are there any security considerations specific to public clouds that need to be taken into account?
- How well do service contract documents communicate the functional scope and capabilities of a service?
- Are there SLA guarantees that need to be tested and verified?
- How easily can the service be composed and recomposed?
- Can the service be moved between on-premise and cloud environments?
- How easily can the service be discovered?
- Is compliance with any industry standards or profiles (such as WS-I profiles) required?
- If cloud deployed, are there proprietary characteristics being imposed by the cloud provider that are not compatible with on-premise service characteristics?
- How effective are the validation rules within the service contract and within the service logic?
- Have all possible service activities and service compositions been mapped out?
- For service compositions that span on-premise and cloud environments, is the performance and behavior consistent and reliable?

Because services are positioned as IT assets with runtime usage requirements comparable to commercial software products, similar quality assurance processes are generally required.

Service Deployment and Maintenance

Service deployment represents the actual implementation of a service into the production environment. This stage can involve numerous interdependent parts of the underlying service architecture and supporting infrastructure, such as:

- Distributed components
- Service contract documents
- Middleware (such as ESB and orchestration platforms)
- Cloud service implementation considerations
- Cloud-based IT resources encompassed by an on-premise or cloud-based service
- Custom service agents and intermediaries
- System agents and processors
- Cloud-based service agents, such as automated scaling listeners and pay-for-use monitors
- On-demand and dynamic scaling and billing configurations
- Proprietary runtime platform extensions
- Administration and monitoring products

Service maintenance refers to upgrades or changes that need to be made to the deployment environment, either as part of the initial implementation or subsequently. It does not pertain to changes that need to be made to the service contract or the service logic, nor does it relate to any changes that need to be made as part of the environment that would constitute a new version of the service.

Service Usage and Monitoring

A service that has been deployed and is actively in use as part of one or more service compositions (or has been made available for usage by service consumers in general) is considered to be in this stage. The ongoing monitoring of the active service generates metrics that are necessary to measure service usage for evolutionary maintenance (such as scalability, reliability, etc.), as well as for business assessment reasons (such as when calculating cost of ownership and ROI).

Special considerations regarding this stage apply to cloud-based services, such as:

- The cloud service may be hosted by virtualized IT resources that are further hosted by physical IT resources shared by multiple cloud consumer organizations.
- The cloud service usage may be monitored not only for performance, but also for billing purposes when its implementation is based on a per-usage

fee license.

- The elasticity of the cloud service may be configured to allow for limited or unlimited scalability, thereby increasing the range of behavior (and changing its usage thresholds) when compared to an on-premise implementation.

This phase is often not documented separately, as it is not directly related to service delivery or projects responsible for delivering or altering services. It is noted in this book because while active and in use, a service can be subject to various governance considerations.

Service Discovery

To ensure that reusable services are consistently reused, project teams carry out a separate and explicitly defined service discovery process. The primary goal of this process is to identify one or more existing agnostic services (such as utility or entity services) within a given service inventory that can fulfill generic requirements for whatever business process the project team is tasked with automating.

The primary mechanism involved in performing service discovery is a service registry that contains relevant metadata about available and upcoming services, as well as pointers to the corresponding service contract documents (which can include SLAs). The communications quality of the metadata and service contract documents play a significant role in how successfully this process can be carried out. This is why the Service Discoverability (300) principle is dedicated solely to ensuring that information published about services is highly interpretable and discoverable.

Service Versioning and Retirement

After a service has been implemented and used in production environments, the need may arise to make changes to the existing service logic or to increase the functional scope of the service. In cases like this, a new version of the service logic and/or the service contract will likely need to be introduced. To ensure that the versioning of a service can be carried out with minimal impact and disruption to service consumers that have already formed dependencies on the service, a formal service versioning process needs to be in place.

There are different versioning strategies, each of which introduces its own set of rules and priorities when it comes to managing the backward and forward compatibilities of services. (Chapter 10 provides fundamental coverage of

common service versioning approaches for Web services and REST services.)

Project Stages and Organizational Roles

Figure 4.31 revisits the SOA project stages and maps them to common organizational roles. These roles are described in the *SOA Governance: Governing Shared Services On-Premise & in the Cloud* text book.

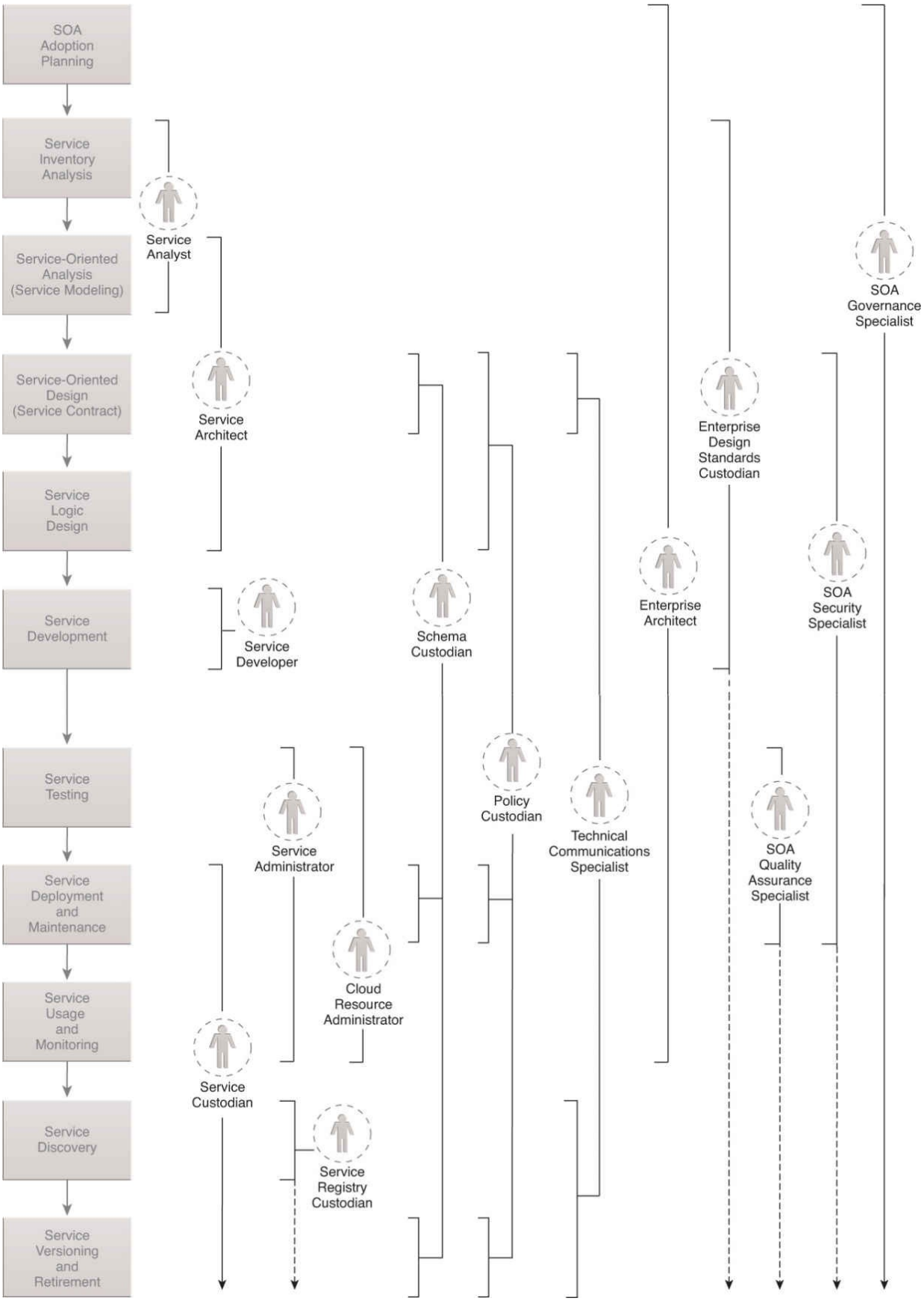
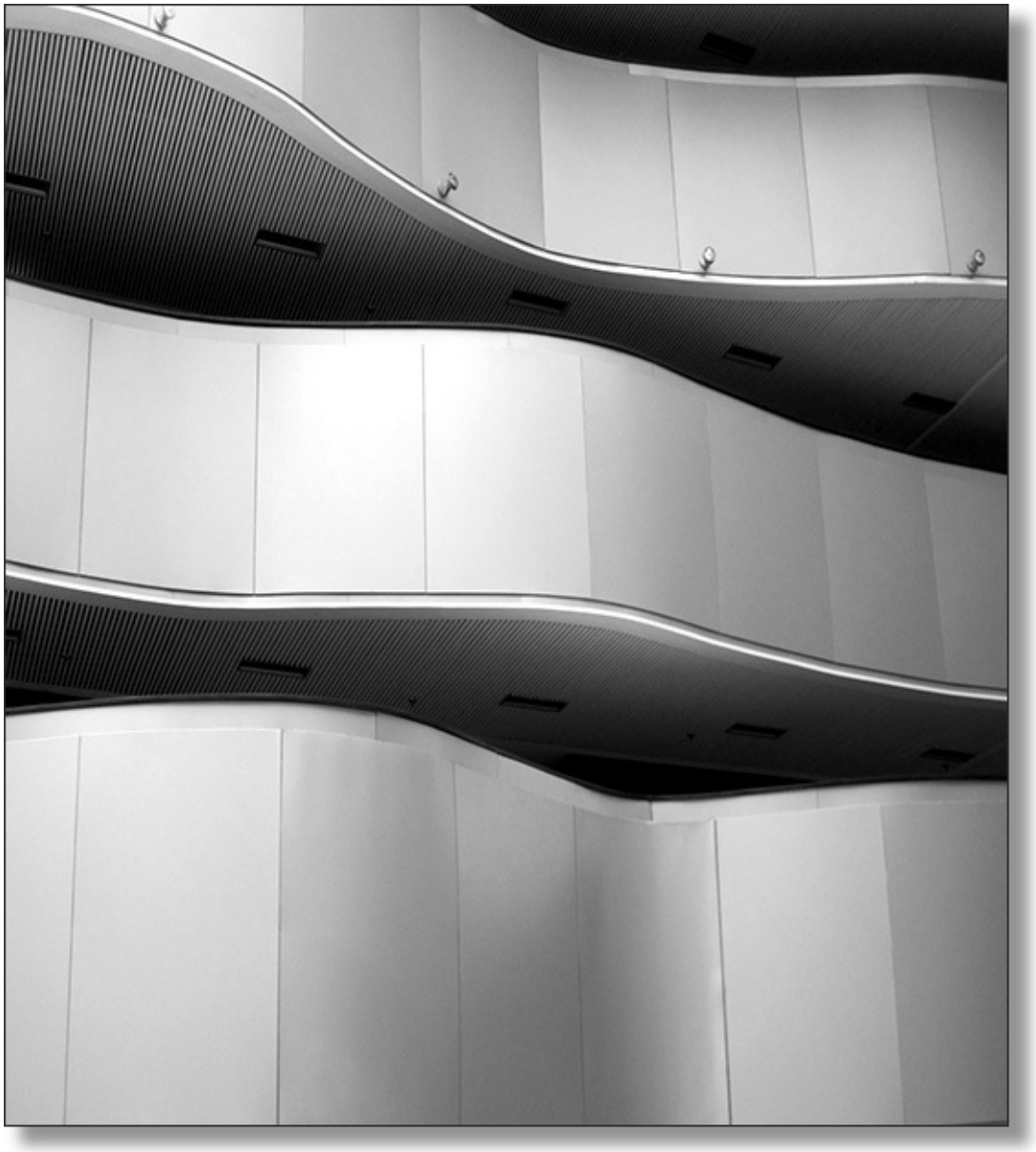

SOA Adoption Planning
Service Inventory Analysis
Service-Oriented Analysis (Service Modeling)
Service-Oriented Design (Service Contract)
Service Logic Design
Service Development
Service Testing
Service Deployment and Maintenance
Service Usage and Monitoring
Service Discovery
Service Versioning and Retirement
Service Analyst
Service Architect
Service Developer
Service Administrator
Cloud Resource Administrator
Service Custodian
Service Registry Custodian
Schema Custodian
Policy Custodian
Technical Communications Specialist
Enterprise Architect
Enterprise Design Standards Custodian
SOA Quality Assurance Specialist
SOA Security Specialist
SOA Governance Specialist

Figure 4.31 Shown here are common associations of organizational roles with different SOA project stages.

Chapter 5. Understanding Layers with Services and Microservices



This chapter provides a concise overview of what lies at the very core of the service-orientation paradigm and the service-oriented architectural model: the

identification and aggregation of agnostic and non-agnostic logic into composable units. These units represent the foundational moving parts that collectively define and enable service-oriented solutions.

The upcoming sections explore this topic area by focusing on a series of primitive process steps, as they are applied to the early stages of service modeling and subsequent service design (Figure 5.1).

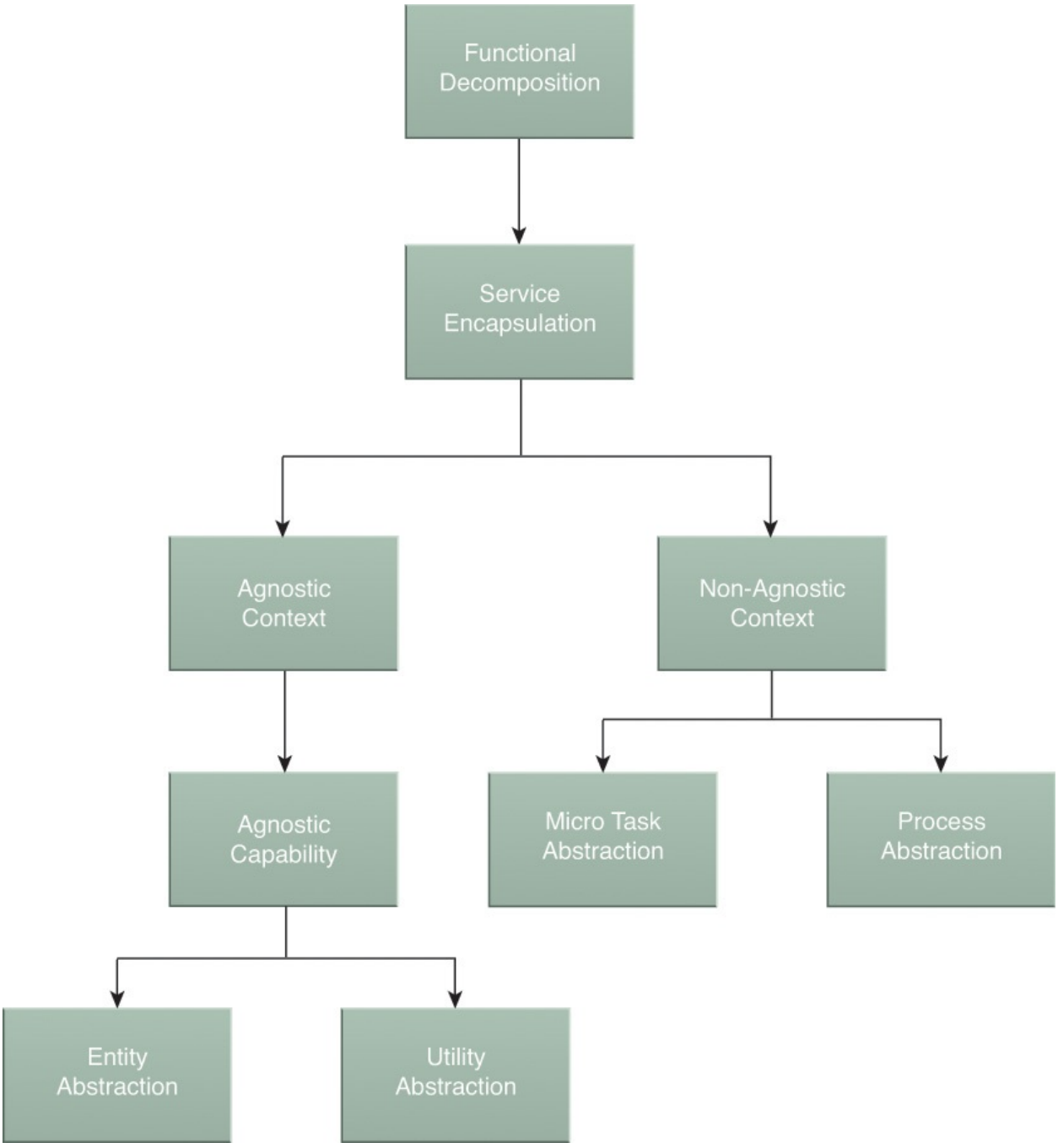


Figure 5.1 A primitive service modeling process that results in the definition of candidate services and capabilities.

5.1 Introduction to Service Layers

The purpose of the service modeling process is essentially to organize a potentially large amount of units of logic so that they can eventually be reassembled into service-oriented solutions. Achieving this requires a set of labels that can be used to group and categorize these units into layers according

to the nature of their logic. The following terms, all of which are referenced in the upcoming sections, help us accomplish this goal.

Service Models and Service Layers

A *service model* is a classification used to indicate that a service belongs to one of several pre-defined types based on the type of logic it contains, the reuse potential of the logic, and how the service may relate to elements of the actual business logic it will help to automate.

The following are common service models:

- *Task Service* – A service with a non-agnostic functional context that generally corresponds to single-purpose, parent business process logic. A task service will usually encapsulate the composition logic required to compose several other services to complete its task.
- *Microservice* – A non-agnostic service often with a small functional scope encompassing logic with specific processing and implementation requirements. Microservice logic is typically not reusable but can have intra-solution reuse potential. The nature of the logic may vary.
- *Entity Service* – A reusable service with an agnostic functional context associated with one or more related business entities (such as invoice, customer, or claim). For example, a Purchase Order service has a functional context associated with the processing of purchase order-related data and logic.
- *Utility Service* – Although a reusable service with an agnostic functional context as well, this type of service is intentionally not derived from business analysis specifications and models. It encapsulates low-level technology-centric functions, such as notification, logging, and security processing.

Note

A variation of the task service model called the *orchestrated task service* performs the same overall function as a task service, but is typically responsible for encompassing extensive orchestration logic, which can involve distinct technologies and middleware. Orchestrated task services are not covered in this book.

Even though a microservice can contain reusable logic, it is considered a non-agnostic service because any reuse potential its logic may have is typically limited to reuse within the parent

business process logic being automated by an application. For a service to be considered agnostic, it must contain logic that is potentially reusable by multiple business processes.

A given service inventory will usually contain multiple services that are grouped based on each of these service models. Each of these groupings is referred to as a *service layer* (Figure 5.2).

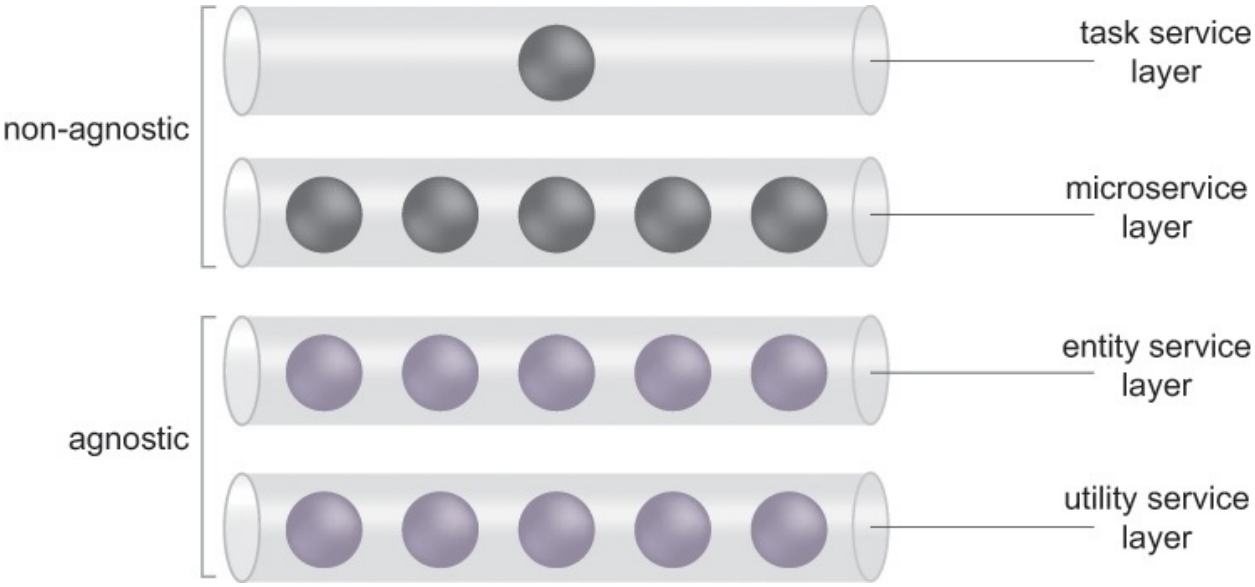


Figure 5.2 The common service layers, each of which is based on a service model.

Service and Service Capability Candidates

The upcoming process is focused on modeling service logic prior to the actual building of the service logic. At this early stage, we are essentially conceptualizing services and their capabilities, which is why qualifying them with the word “candidate” is helpful. The terms “service candidate” and “service capability candidate” are used to distinguish conceptualized service logic from service logic that has already been implemented. This distinction is important, particularly because candidate service logic that has not yet been conceptualized may be subject to further practical considerations that may result in additional changes during service design and development.

5.2 Breaking Down the Business Problem

The typical starting point is termed a “business problem,” which can be any business task or process for which an automation solution is required. To apply

service-orientation, we first must break down a business process by functionally decomposing it into a set of granular actions. This enables us to identify potential functional contexts and boundaries that may become the basis of services and service capabilities. During this initial decomposition stage, we focus primarily on organizing business process actions into two primary categories: agnostic and non-agnostic.

Functional Decomposition

The separation of concerns theory is based on an established software engineering principle that promotes the decomposition of a larger problem into smaller problems (called “concerns”) for which corresponding units of solution logic can be built. The rationale is that a larger problem, such as the execution of a business process, can be more easily and effectively solved when separated into smaller parts. Each unit of solution logic that is built exists as a separate body of logic that is responsible for solving one or more of the identified smaller concerns (Figure 5.3). This design approach forms the basis for distributed computing.

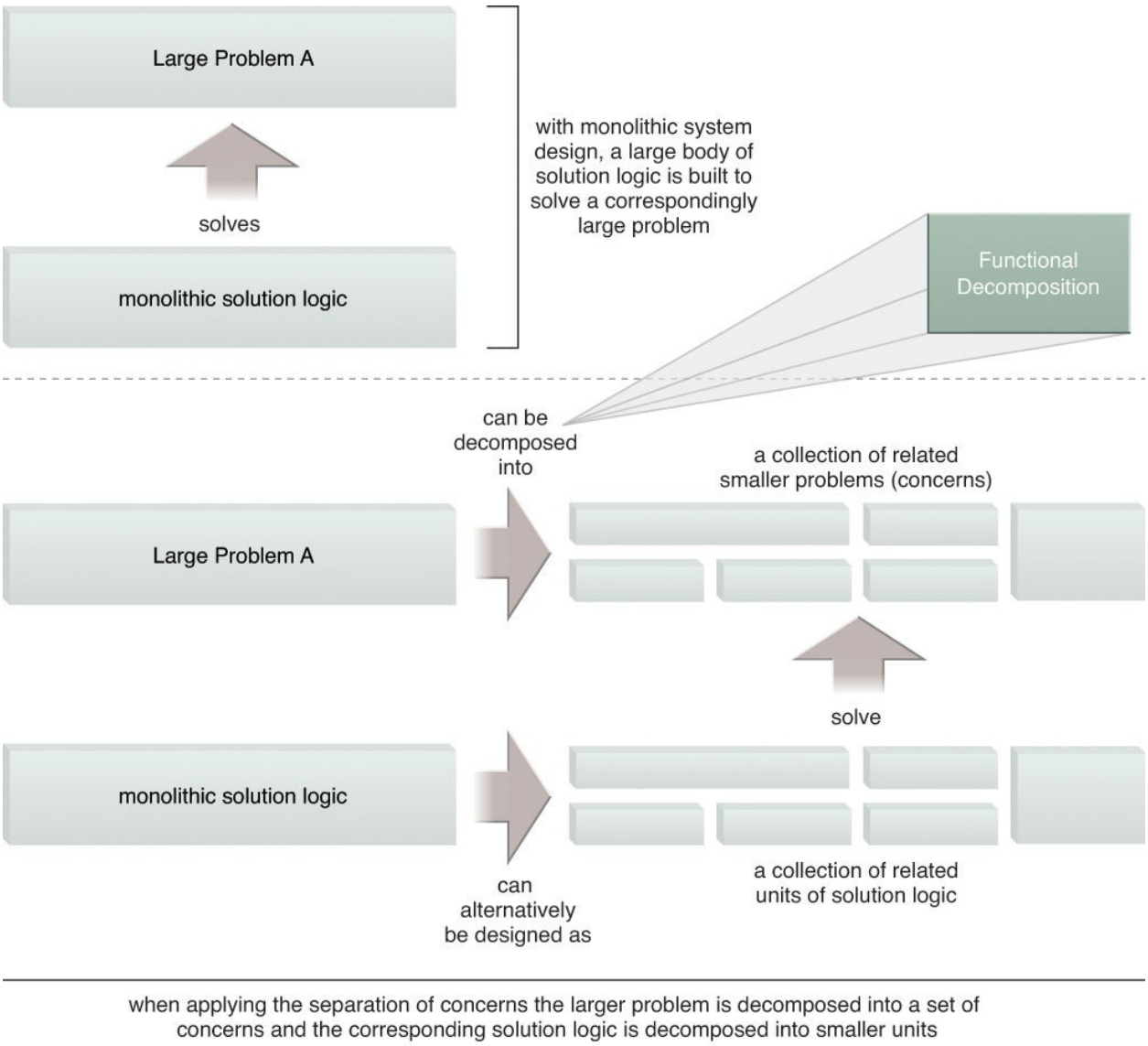


Figure 5.3 A larger problem is decomposed into multiple, smaller problems. Later steps focus on the definition of solution logic units that individually address these smaller problems.

Service Encapsulation

When assessing the individual units of solution logic that are required to solve a larger problem, we may realize that only a subset of the logic is suitable for encapsulation within services. During the service encapsulation step, we identify the parts of the logic required that are suitable for encapsulation by services (Figure 5.4).

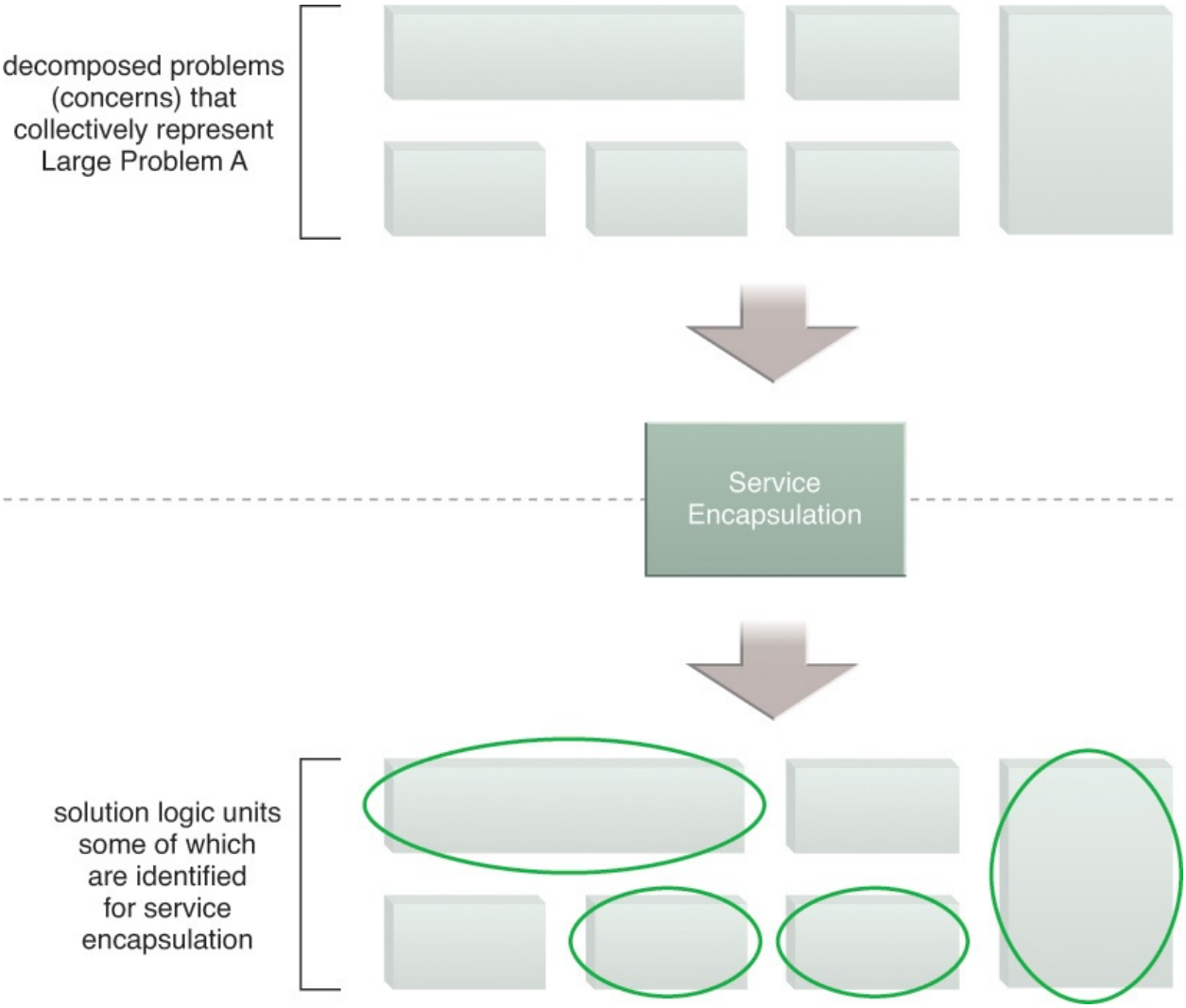


Figure 5.4 Some of the decomposed solution logic is identified as being not suitable for service encapsulation. The highlighted blocks represent logic that is deemed suitable for encapsulation by services.

Agnostic Context

After the initial decomposition of solution logic, we will typically end up with a series of solution logic units that correspond to specific concerns. Although some of this logic may be capable of solving other concerns, grouping single-purpose and multipurpose logic together prevents us from being able to realize any potential reuse. By identifying the parts of this logic that are not specific to known concerns, we are able to separate and reorganize the appropriate logic into a set of agnostic contexts (Figure 5.5).

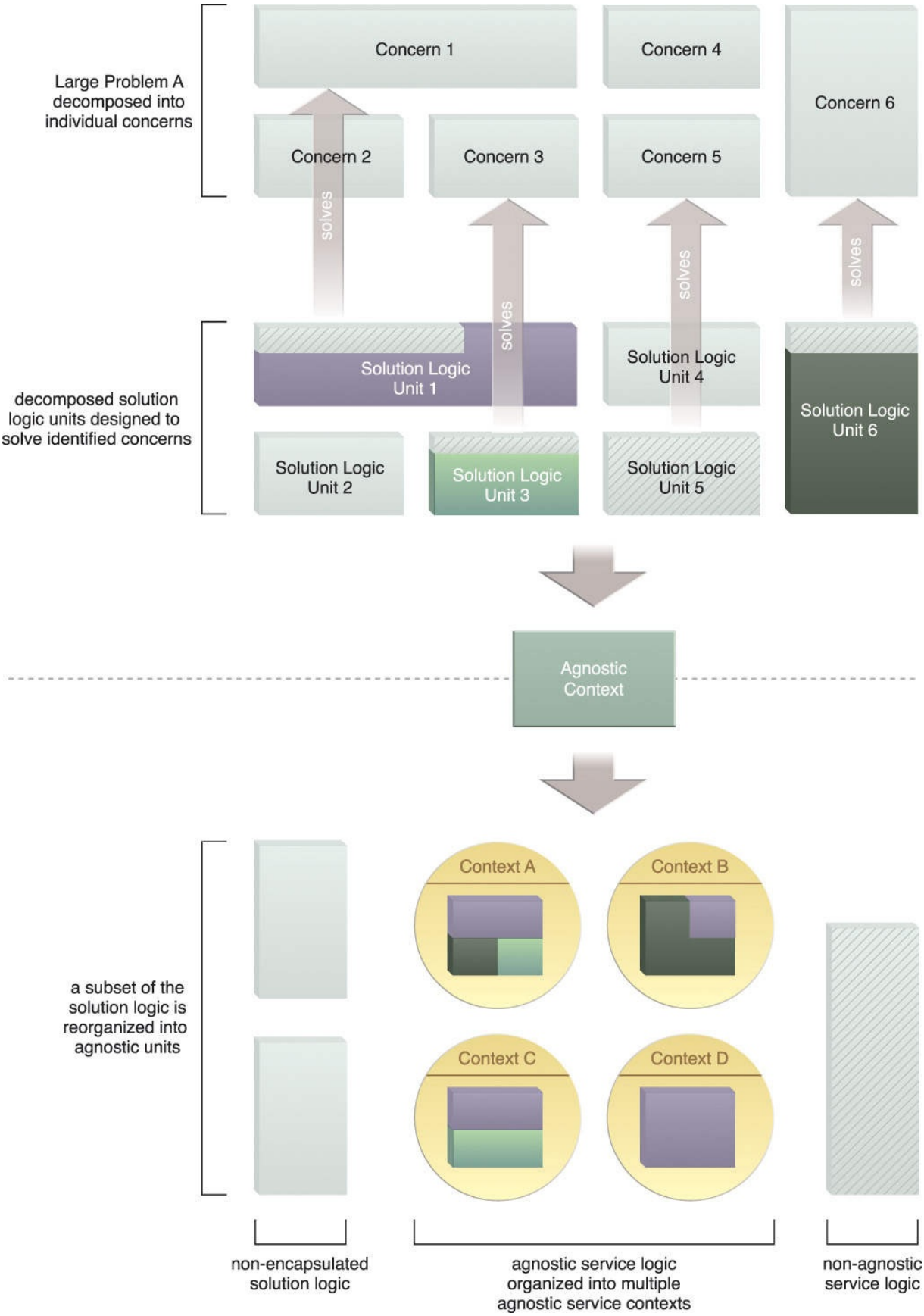

Large Problem A decomposed into individual concerns
Concern 1
Concern 2
Concern 3
Concern 4
Concern 5
Concern 6
solves
solves
solves
solves
decomposed solution logic units designed to solve identified concerns
Solution Logic Unit 1
Solution Logic Unit 2
Solution Logic Unit 3
Solution Logic Unit 4
Solution Logic Unit 5
Solution Logic Unit 6
Agnostic Context
a subset of the solution logic is reorganized into agnostic units
Context A
Context B
Context C
Context D
non-encapsulated solution logic
agnostic service logic organized into multiple agnostic service contexts
non-agnostic service logic

Figure 5.5 Decomposed units of solution logic will naturally be designed to solve concerns specific to a single, larger problem. Solution Logic Units 1, 3, and 6 represent logic that contains multipurpose functionality trapped within a single-purpose (single concern) context. This step results in a subset of the solution logic being further decomposed and distributed into services with specific agnostic contexts.

Agnostic Capability

Within each agnostic service context, the logic is further organized into a set of agnostic service capabilities. It is, in fact, the service capabilities that address individual concerns. Because they are agnostic, the capabilities are multipurpose and can be reused to solve multiple concerns (Figure 5.6).

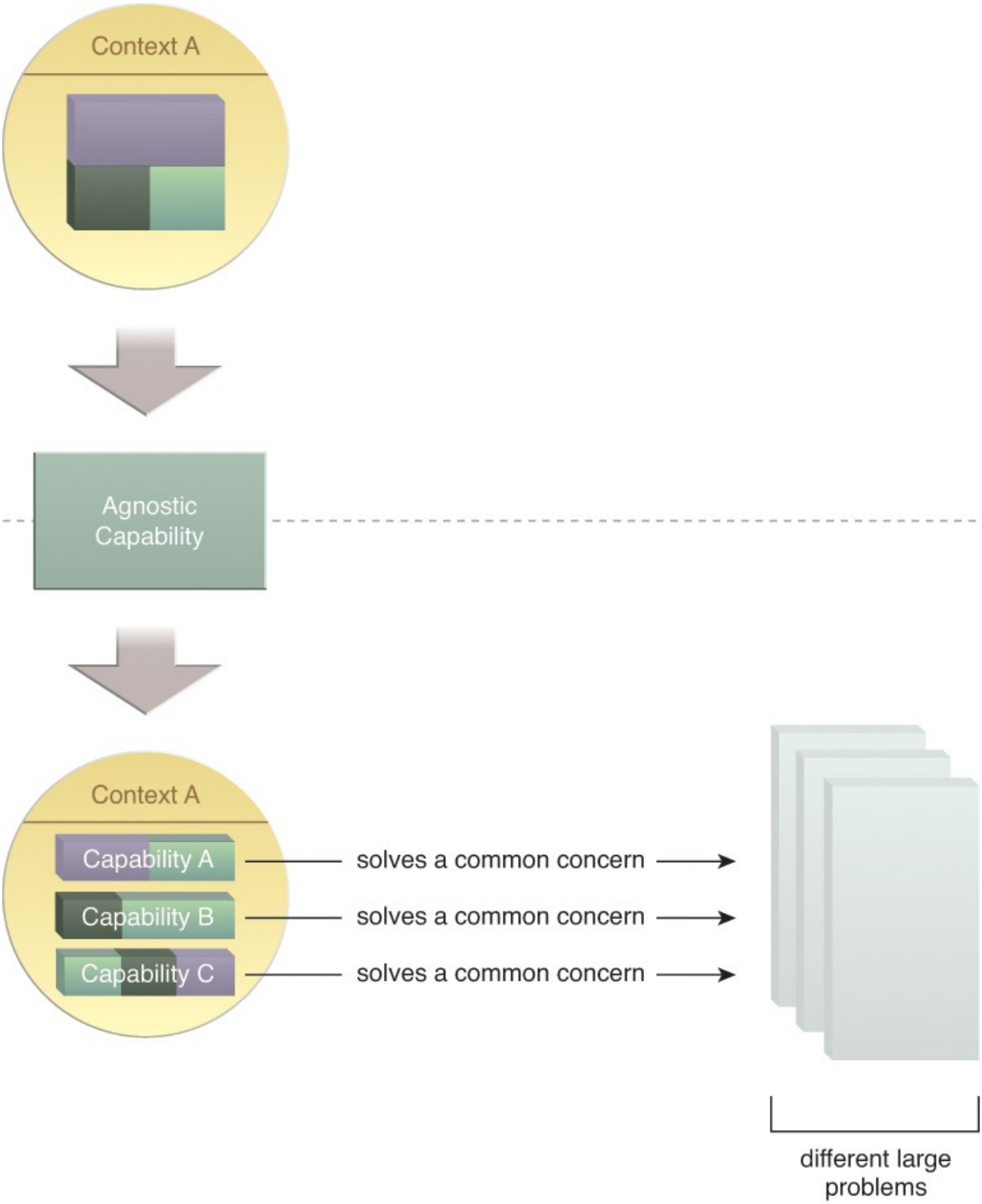


Figure 5.6 A set of agnostic service capabilities is defined, each capable of solving a common concern.

Utility Abstraction

The next step is to separate common, cross-cutting functionality that is neither specific to a business process nor a business entity. This establishes a specialized agnostic functional context limited to logic that corresponds to the utility service model. Repeating this step within a service inventory can result in the creation of multiple utility service candidates and, consequently, a logical utility service layer (Figure 5.7).

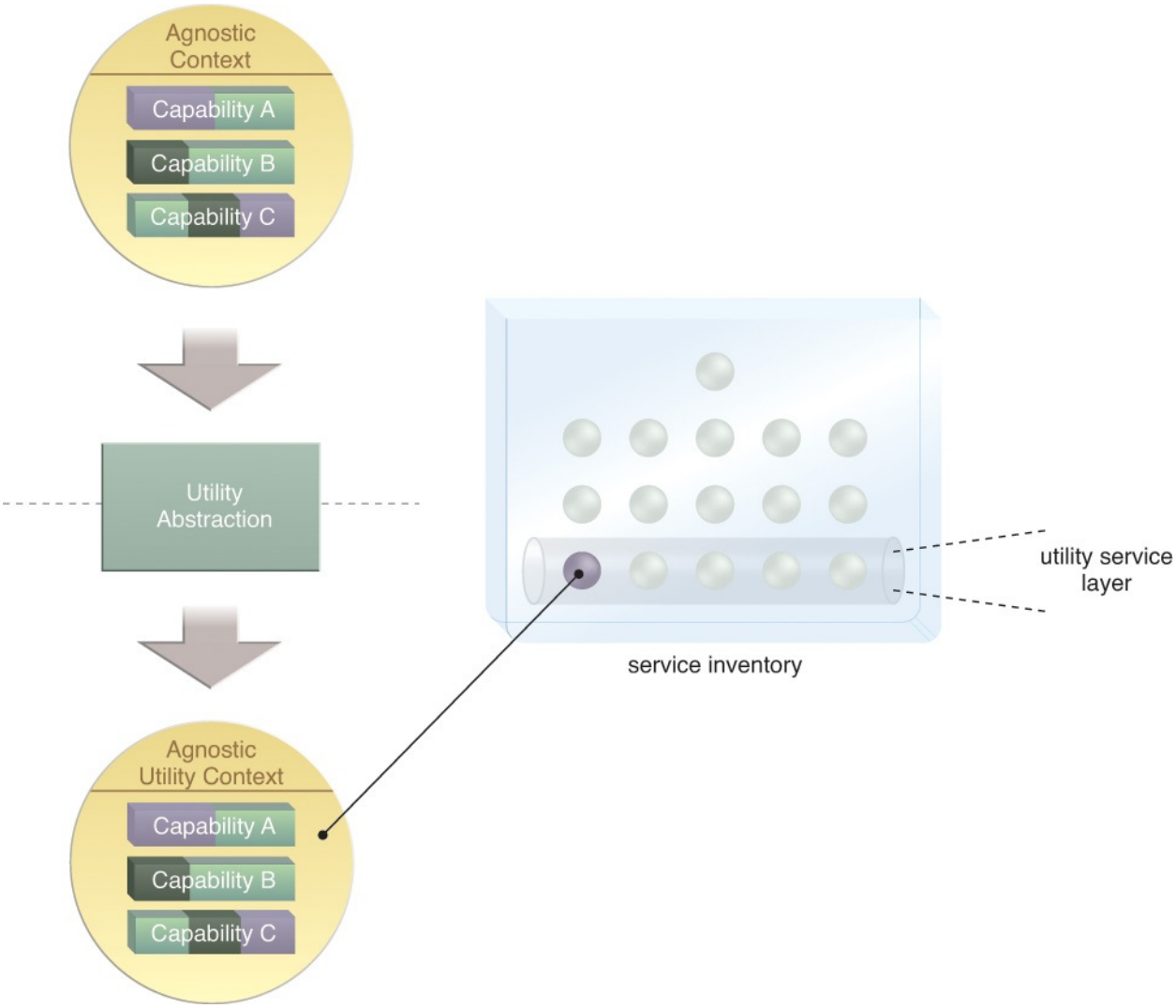


Figure 5.7 Utility-centric agnostic service logic is organized into a utility service layer.

Entity Abstraction

Every organization has business entities that represent key artifacts relevant to how operational activities are carried out. This step is focused on shaping the functional context of a service so that it is limited to logic that pertains to one or more related business entities. As with utility abstraction, repeating this step

tends to establish its own logical service layer (Figure 5.8).

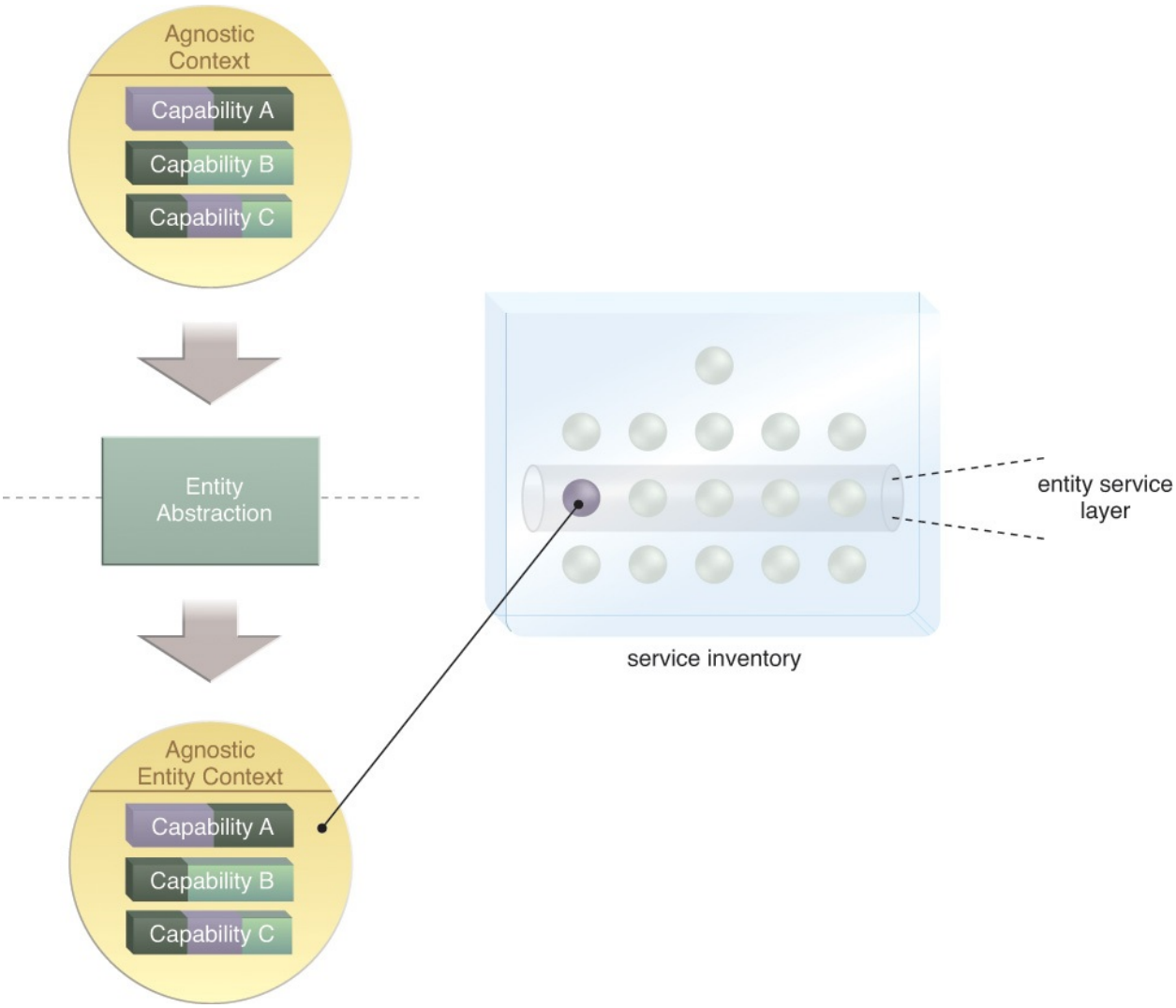


Figure 5.8 Entity-centric agnostic service logic is organized into an entity service layer.

Non-Agnostic Context

The fundamental service identification and definition effort detailed so far has focused on the separation of multipurpose, or agnostic, service logic. What remains after the multipurpose logic has been separated is logic that is specific to the business process. Because this logic is considered single-purpose in nature, it is classified as non-agnostic (Figure 5.9).

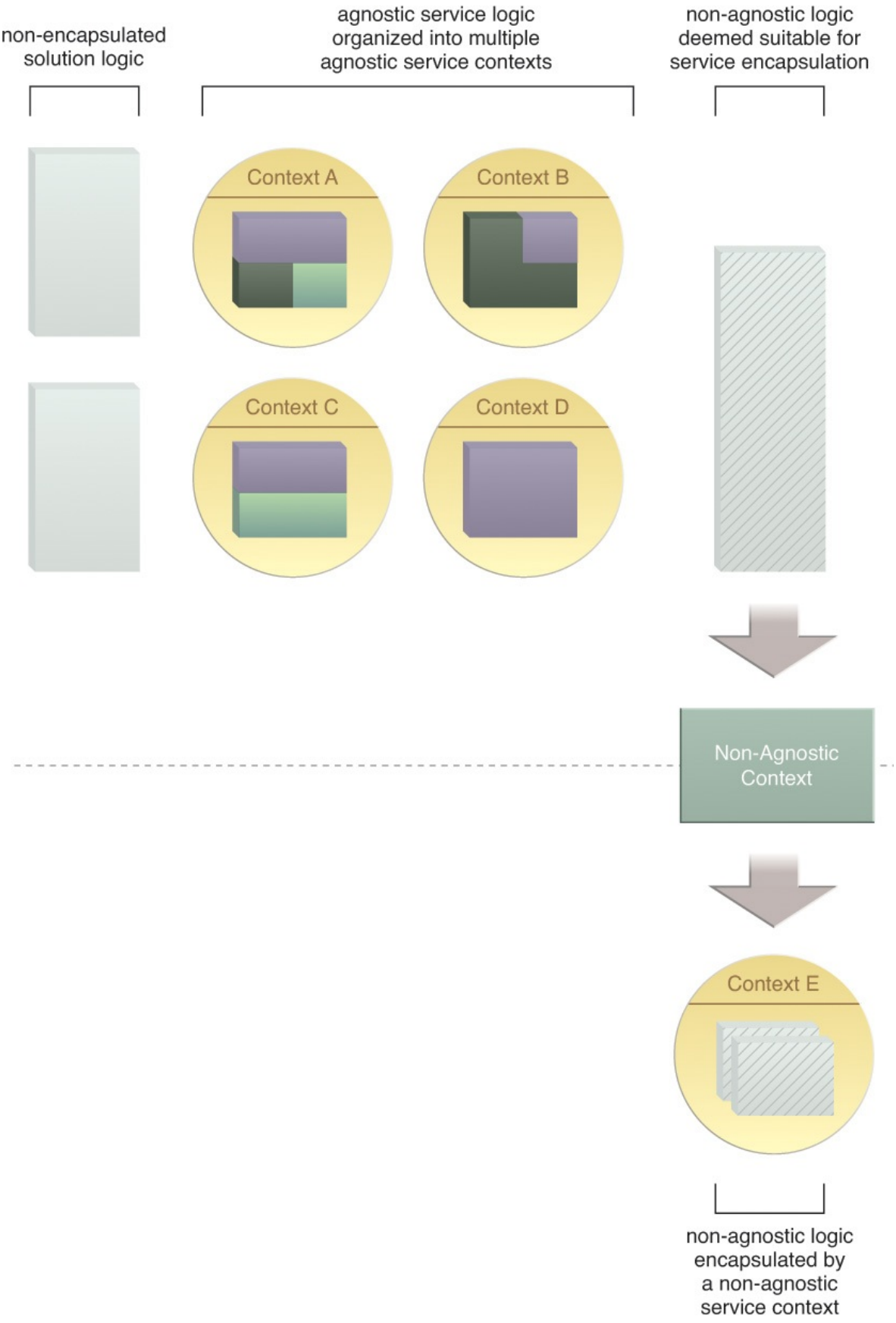

non-encapsulated solution logic
agnostic service logic organized into multiple agnostic service contexts
non-agnostic logic deemed suitable for service encapsulation
Context A
Context B
Context C
Context D
Non-Agnostic Context
Context E
non-agnostic logic encapsulated by a non-agnostic service context

Figure 5.9 By revisiting the decomposition process, the remaining service logic can now be categorized as non-agnostic.

Micro Task Abstraction and Microservices

When reviewing available non-agnostic logic, it can become evident that subsets of this logic (or “micro tasks”) may have specific performance or reliability requirements. This type of processing logic can be abstracted into a separate service layer that can benefit from the distinct implementation characteristics of microservices (Figure 5.10).

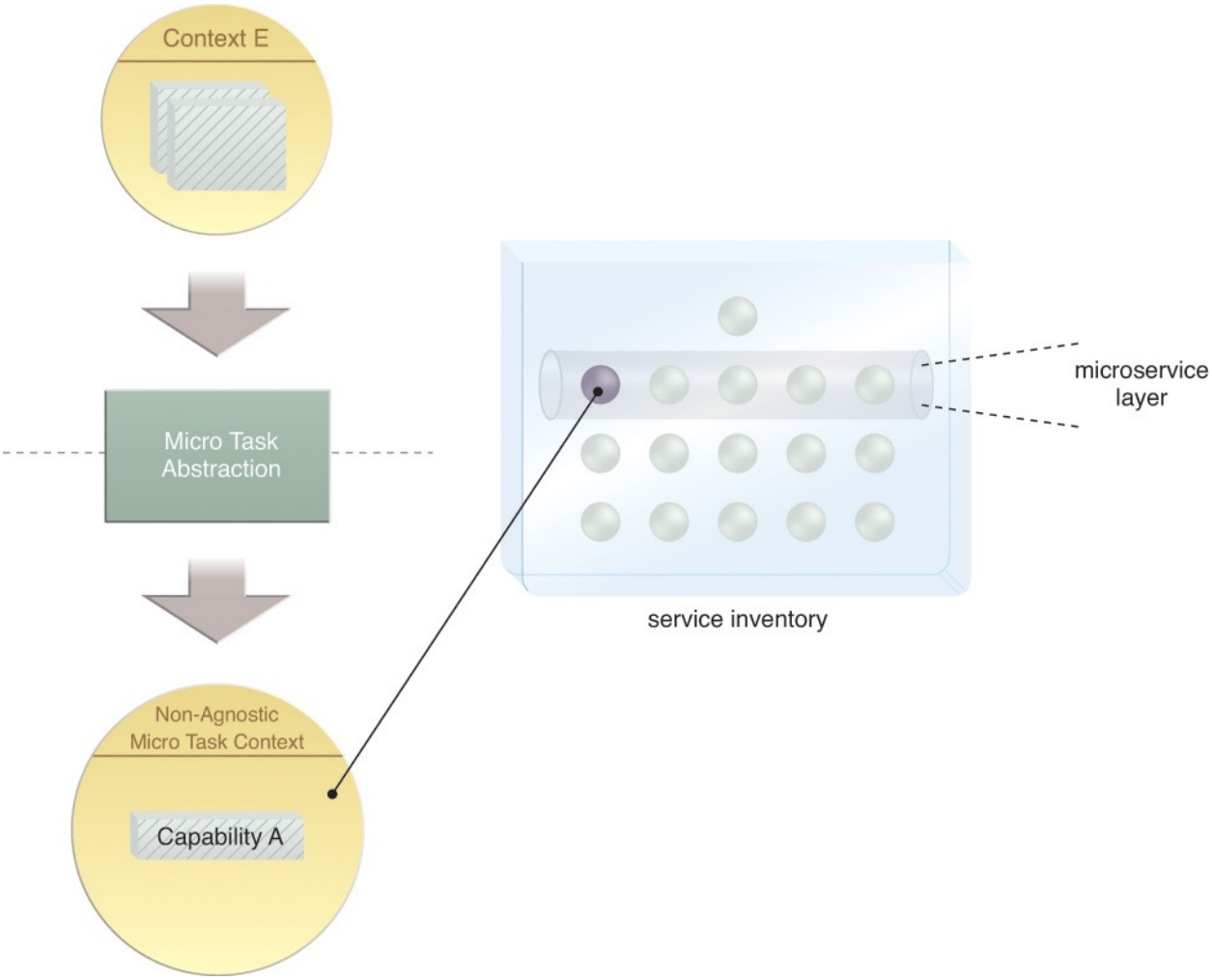


Figure 5.10 Select non-agnostic logic is separated into microservice candidates.

Process Abstraction and Task Services

Abstracting the remaining business process-specific logic into its own service layer will typically result in the creation of a task service, the scope of which is

generally limited to the parent business process (Figure 5.11). The types of logic that are generally encapsulated by a task service are decision logic, composition logic, and other forms of logic that are unique to the business process they are responsible for automating. This responsibility generally puts the task service in control of the execution of an entire service composition, a role known as the *composition controller*.

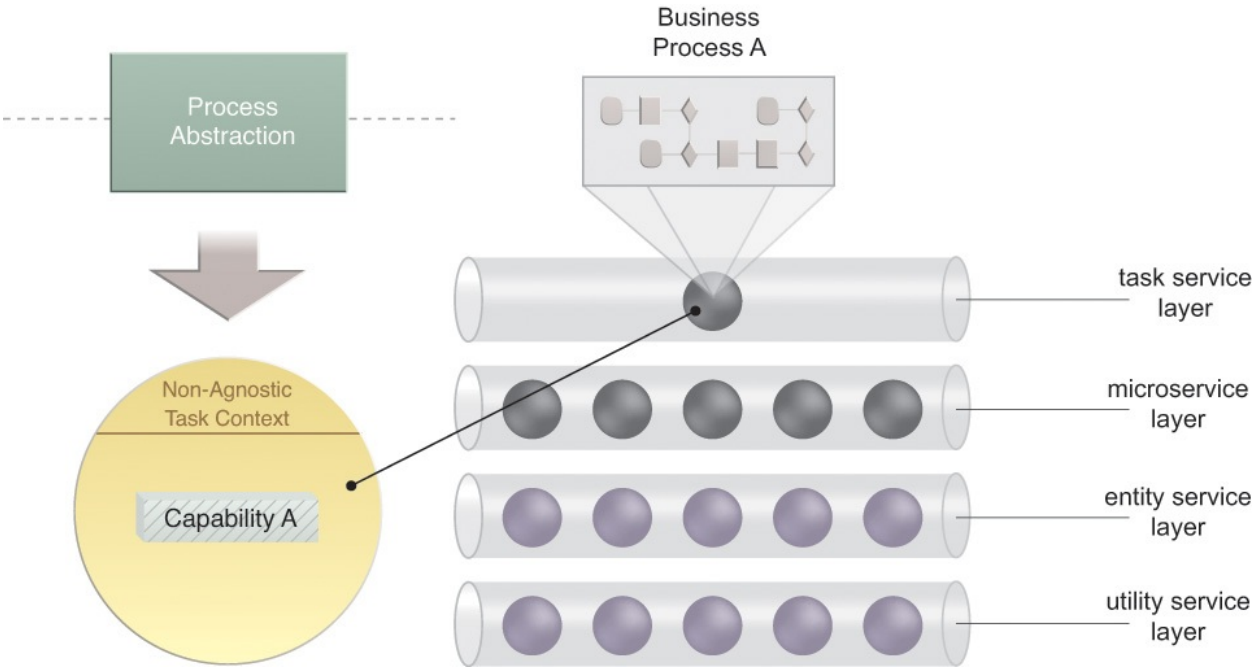


Figure 5.11 The task service represents a part of a parent service layer and is responsible for encapsulating the remaining logic specific to the parent business process.

5.3 Building Up the Service-Oriented Solution

One of the fundamental characteristics that distinguishes service-oriented technology architecture from other forms of distributed architecture is composition-centricity, meaning there is a baseline requirement to inherently support both the composition and *recomposition* of the moving parts comprising a given solution.

In this section, we cover several key aspects of composition in relation to service-orientation, before continuing with the process steps in order to reassemble the logic that has been decomposed in the preceding steps.

Service-Orientation and Service Composition

A baseline requirement for achieving the strategic goals of service-oriented

computing is that those services classified as agnostic be inherently composable. As a means of realizing these goals, the service-orientation design paradigm is naturally focused on enabling flexible composition.

This dynamic is illustrated in Figure 5.12, where we can see how the collective application of service-orientation principles shapes software programs into services that are essentially "composition-ready," meaning they are interoperable, compatible, and composable with other services belonging to the same service inventory.

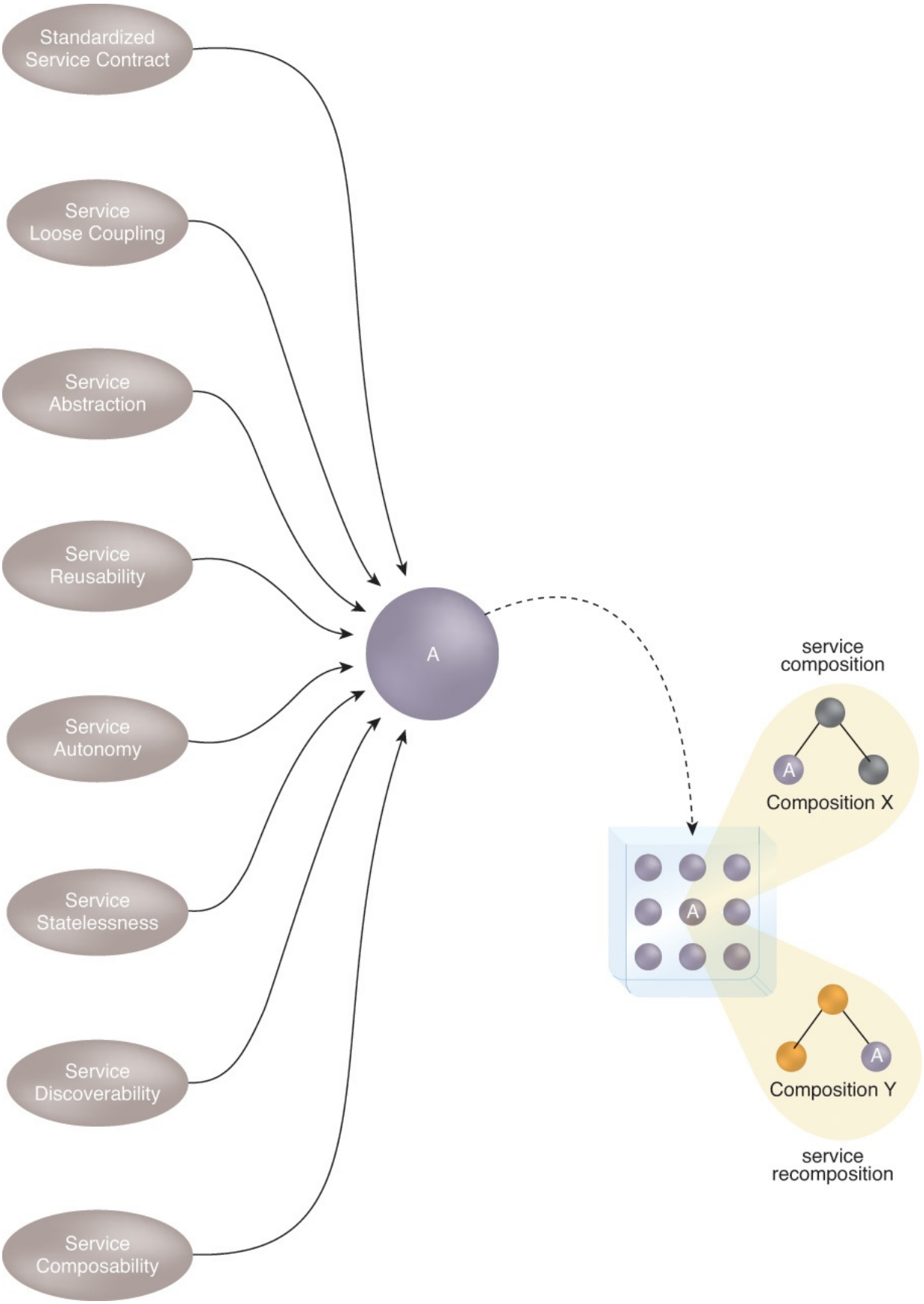

Standardized Service Contract
Service Loose Coupling
Service Abstraction
Service Reusability
Service Autonomy
Service Statelessness
Service Discoverability
Service Composability
A
service composition
A
Composition X
A
A
Composition Y
service recomposition

Figure 5.12 Service A (middle) is a software program shaped into a unit of service-oriented logic by the application of service-orientation design principles. Service A is delivered within a service inventory that contains a collection of services to which service-orientation principles were also applied. The result is that Service A can participate initially in Composition X and, more importantly, can later be pulled into Composition Y and additional service compositions as required.

Figure 5.12 does not only illustrate the aggregation that services can participate in. All distributed systems are comprised of aggregated software programs. What is fundamentally distinct about how service-orientation positions agnostic services is that they are *repeatedly composable*, allowing for subsequent recomposition.

This is what lies at the core of realizing organizational agility as a primary goal of adopting service-oriented computing. Ensuring that a set of services (within the scope determined by the service inventory) is naturally interoperable and designed for participation in complex service compositions enables us to fulfill new business requirements and automate new business processes (Figure 5.13), by augmenting existing service compositions or creating new service compositions with reduced effort and expense. This target state is what leads to the Reduced IT Burden goal of service-oriented computing.

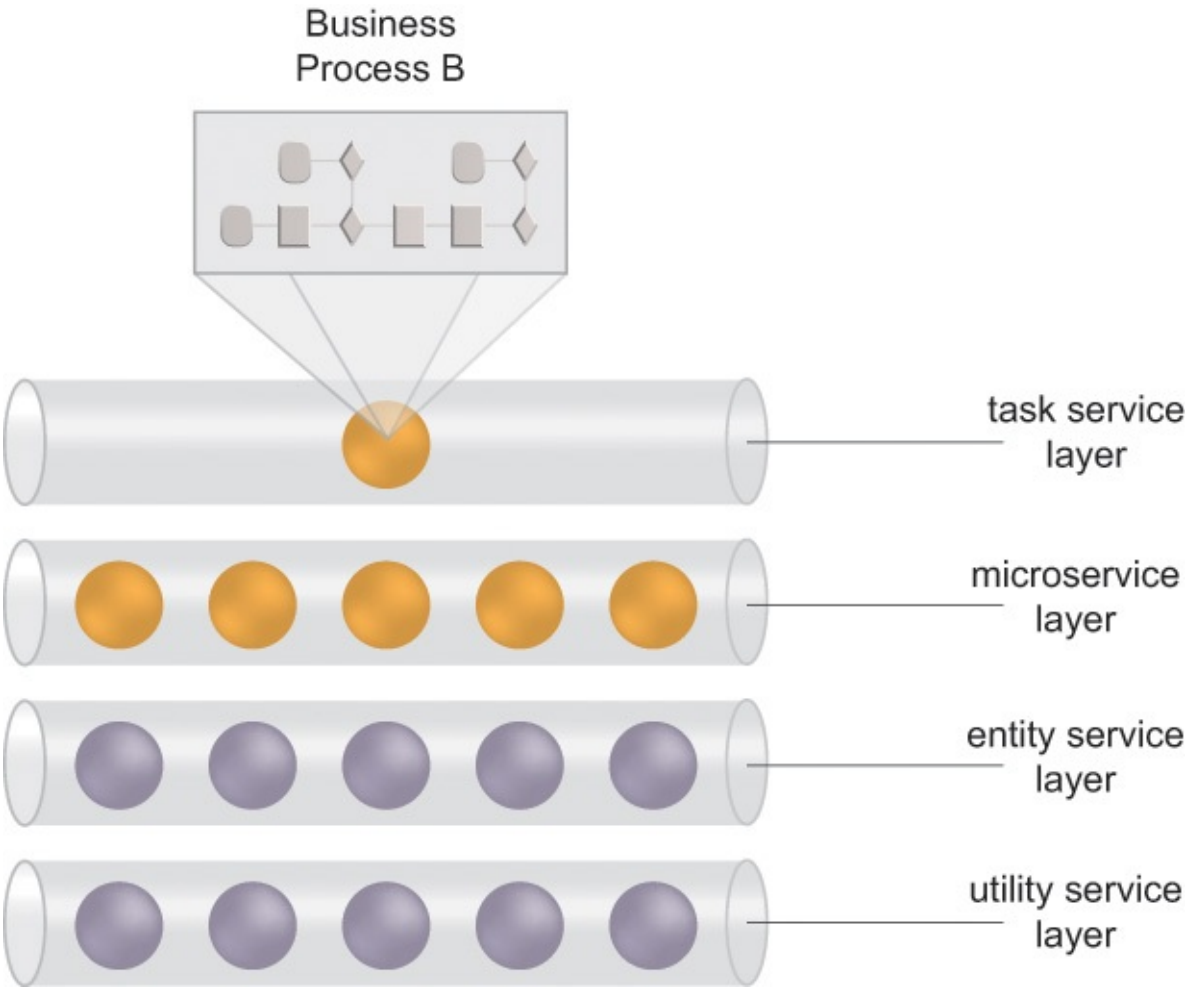


Figure 5.13 The same entity and utility service layers from before, now available for composition by a different set of non-agnostic service candidates in support of the automation of a new business process.

Among the eight service-orientation design principles, one is specifically relevant to service composition design. The Service Composability principle is solely dedicated to shaping a service into an effective composition participant. All other principles support Service Composability in achieving this objective (Figure 5.14). In fact, as a regulatory principle, Service Composability is applied primarily by ensuring that the design goals of the other seven principles are realized to a sufficient degree.

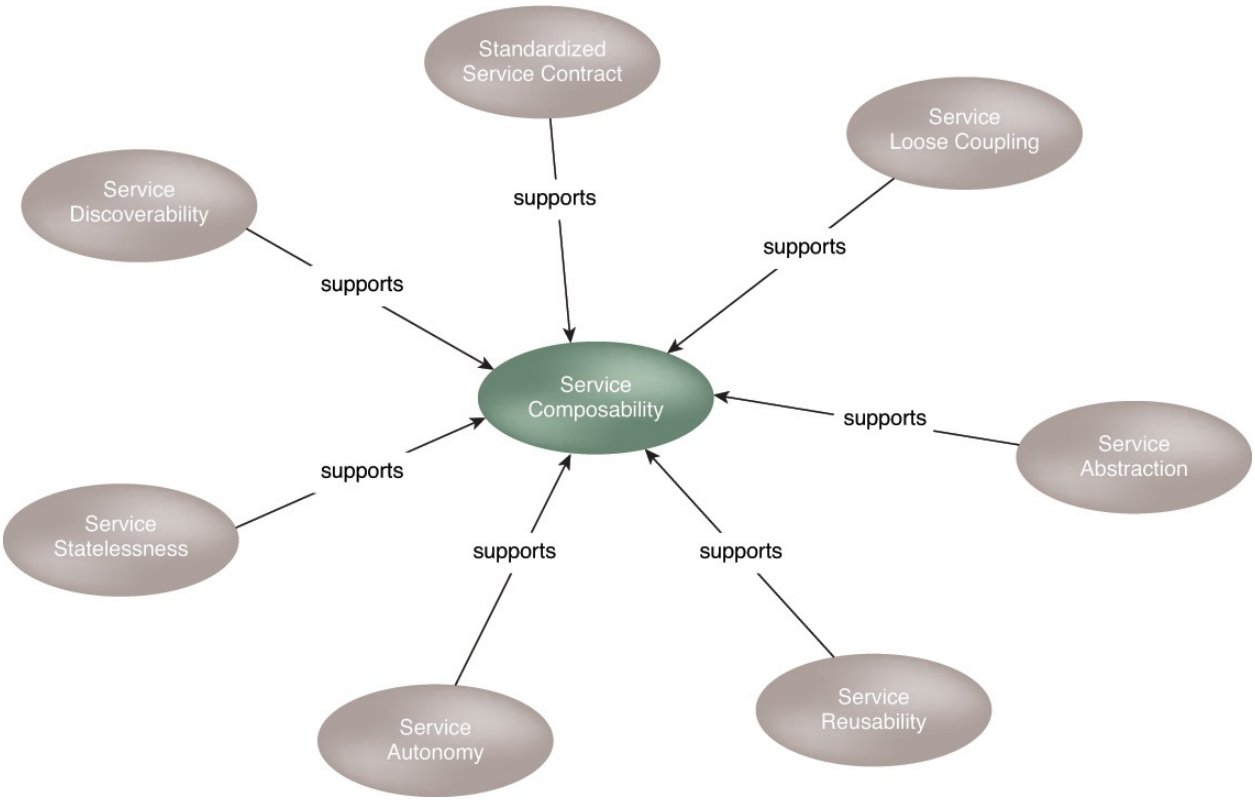


Figure 5.14 A common objective of all service-orientation design principles is the shaping of services in support of increased composability potential.

Capability Composition and Capability Recomposition

Up until now in the process steps, logic has only been separated into individual functional contexts and capabilities. This provides us with a pool of well-defined building blocks from which we can assemble automation solutions. The steps that follow are focused on carrying out this building process via the composition and recomposition of service capability candidates (Figure 5.15).

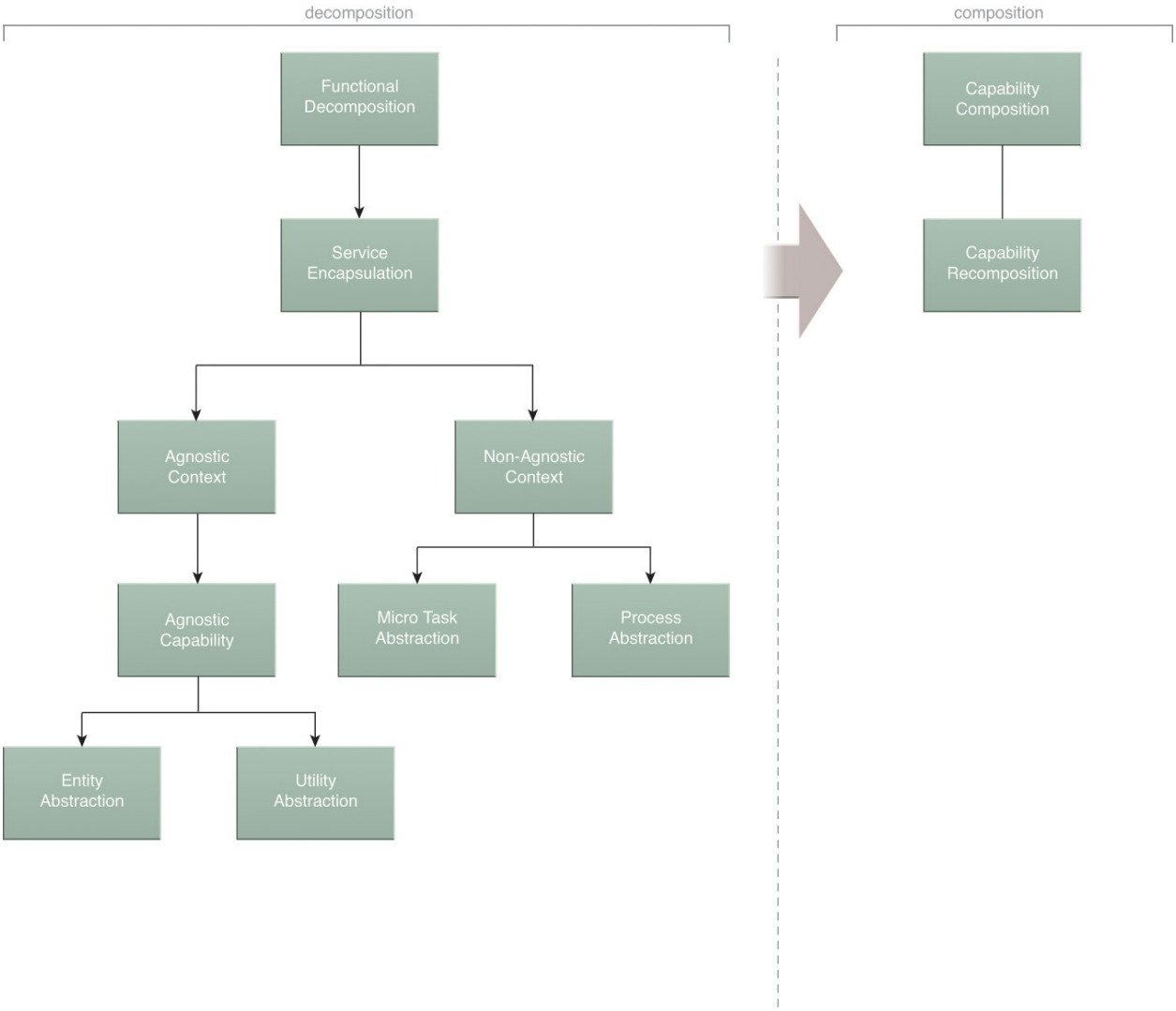


Figure 5.15 Subsequent to the decomposition of a business problem into units of service logic, we focus on how these units can be assembled into service-oriented solutions.

Capability Composition

Candidate service capabilities are sequenced together in order to assemble the decomposed service logic into a specific service composition that is capable of solving a specific larger problem (Figure 5.16). Much of the logic that determines which service capabilities to invoke and in which order they are to be composed will usually reside within the task service.

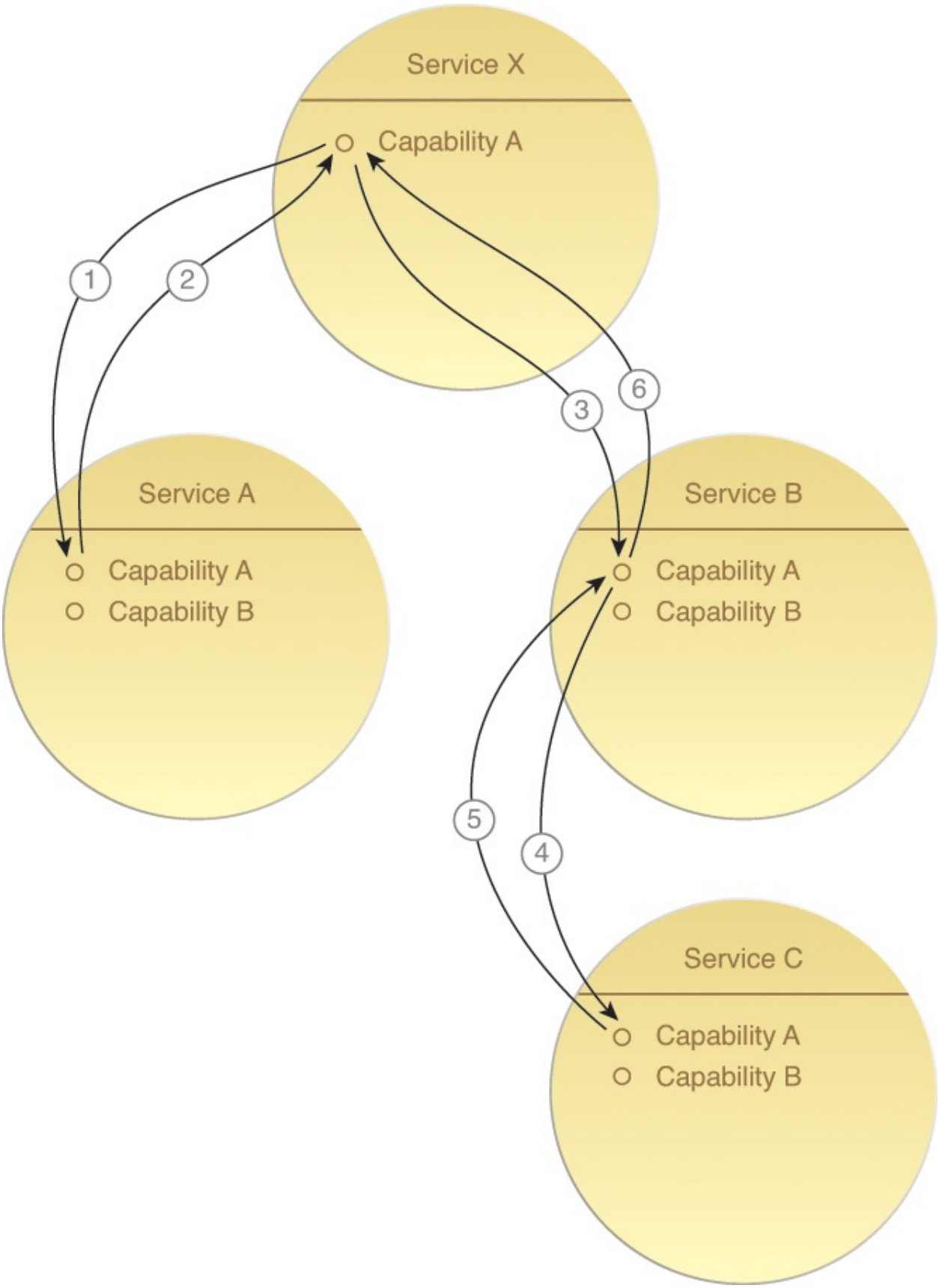

Service X
Capability A
1
2
3
6
Service A
Capability A
Capability B
Service B
Capability A
Capability B
5
4
Service C
Capability A
Capability B

Figure 5.16 Although generally referred to as a service composition, services that compose each other actually do so via their individual service capabilities.

Beyond forming the basis for the basic aggregation of service functionality, this step reinforces functional service boundaries by requiring a service that needs access to logic outside of its context to access this logic via the composition of another service. This requirement avoids redundancy of logic across services.

Capability Composition and Microservices

The type of logic placed in microservices will generally have specific performance and/or reliability requirements. The microservice model can therefore introduce the need for a distinct implementation environment optimized to support special processing demands. Microservice implementations are often highly autonomous in order to minimize dependencies on resources outside of their functional boundaries that could compromise fulfilling their processing requirements.

As a result, when a microservice needs to access other resources, those resources can either be replicated or redundantly implemented so that they remain part of the microservice's local processing scope. Therefore, when it is decided that a microservice needs to compose another service, the composed service may be redundantly implemented and deployed together with the microservice.

Let's imagine that Service B in Figure 5.16 is a microservice and Service C is a utility service being composed by the microservice. The logical view provided by Figure 5.16 would stay the same. However, the physical view of this composition architecture could vary, depending on what technologies are utilized as part of the microservice implementation environment. For example, Figure 5.17 shows how both the microservice and utility service could be rolled out in the same deployment bundle and placed onto a dedicated virtual server. Figure 5.18 takes this a step further by physically grouping the services together with system files and libraries within a container. In either architecture, that same utility service may be in use in various other capacities, within this and other solutions, but it is specifically redundantly deployed in support of the one microservice.

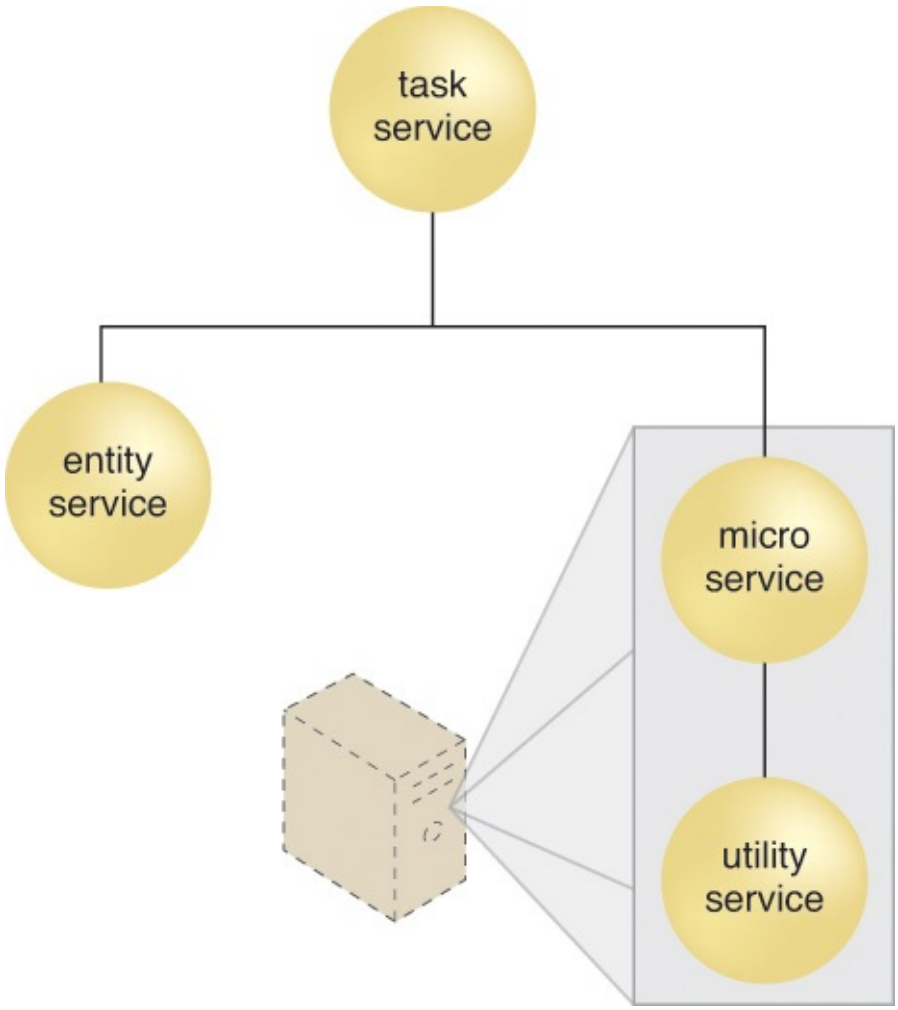


Figure 5.17 The microservice and a redundant implementation of the utility service it is composing are grouped in the same deployment bundle and located on a dedicated virtual server. This increases the autonomy of the microservice, which it may need to fulfill its specialized processing requirements.

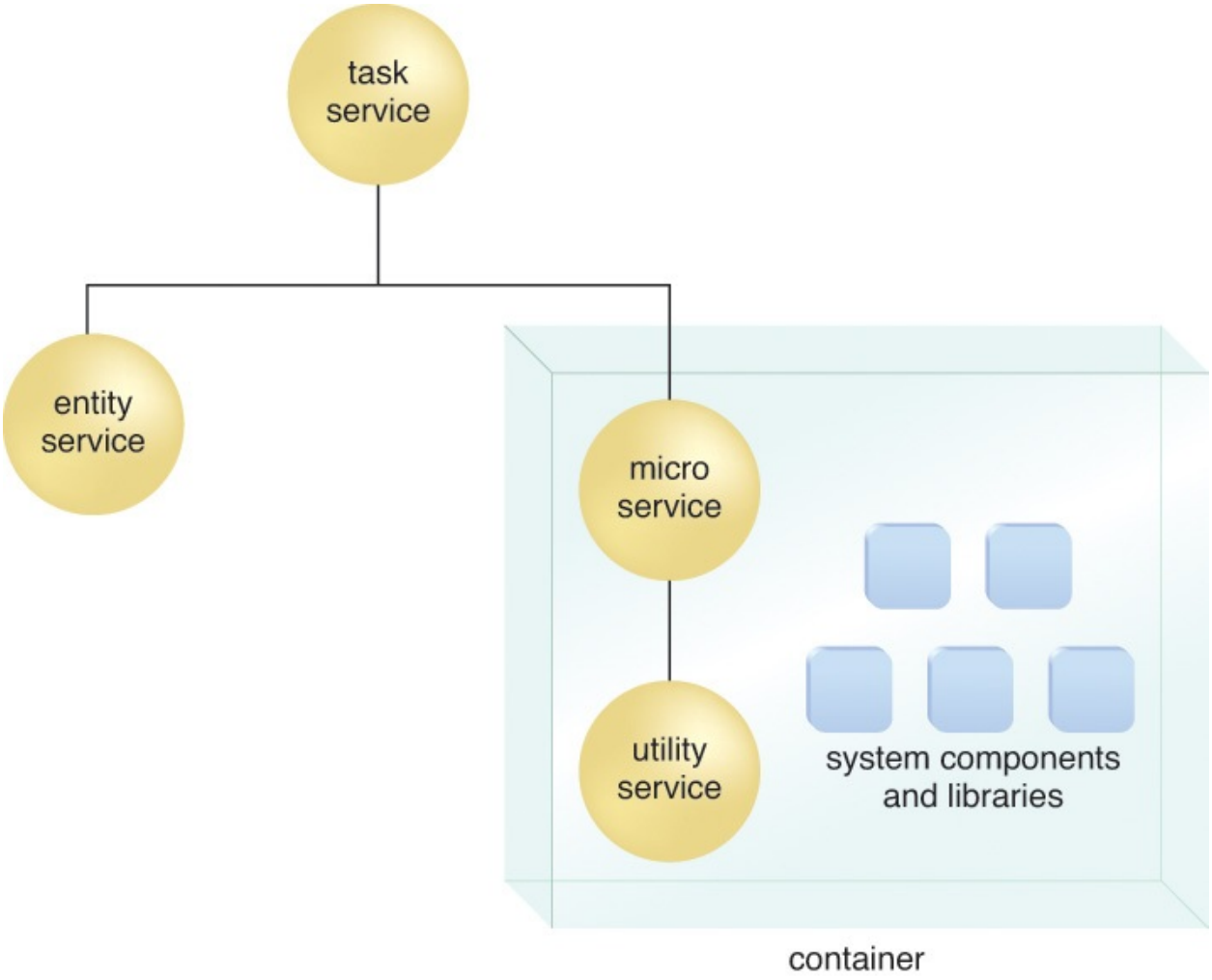


Figure 5.18 The microservice and the redundant implementation of the utility service are positioned within a container that also includes system components and libraries. This is an example of how containerization technology can be used to further increase the autonomy and mobility of services. The extent to which autonomy is increased depends on the extent to which redundant implementations of external resources the service may need to call are included in the container.

Note that Figures 5.17 and 5.18 depict architectures that are commonly associated with microservice implementations. Deployment bundles and containerization technology can also be used for services based on other service models or for entire solutions that are not service-oriented. Due to the typical requirement of a microservice to support specialized processing or deployment requirements, there is usually a greater need for dedicated underlying hosting environments and resources.

Numerous variations of these architectures can exist. For example:

- Services packaged in the same deployment bundle may be able to communicate in-process or out-of-process.
- The microservice in the preceding scenarios may compose the utility service to access an underlying resource or it may disregard the Service Loose Coupling principle and access the underlying resource directly.
- Multiple deployment bundles can be located on the same virtual server, as long as respective autonomy requirements can be fulfilled.
- In Figure 5.18, the container is located on a physical server, but it can also be located on a virtual server.
- A container can host multiple deployment bundles, which may be desirable if communication between services and resources in the respective bundles is required.

Although microservice architecture and related technologies are not covered in this book, summary profiles of the Microservice Deployment [349] and Containerization [333] patterns are provided in Appendix C and are recommended reading. These and other related patterns can also be accessed in the *Service Implementation Patterns* category at www.soapatterrns.org.

Capability Recomposition

As previously mentioned, the recomposition of services is a fundamental and distinctive goal of service-oriented computing. This step specifically addresses the recurring involvement of a service via the repeated composition of a service capability. The relationship diagram shown in Figure 5.19 highlights how the preceding steps that have been described all essentially lead to opportunities for service capability recomposition.

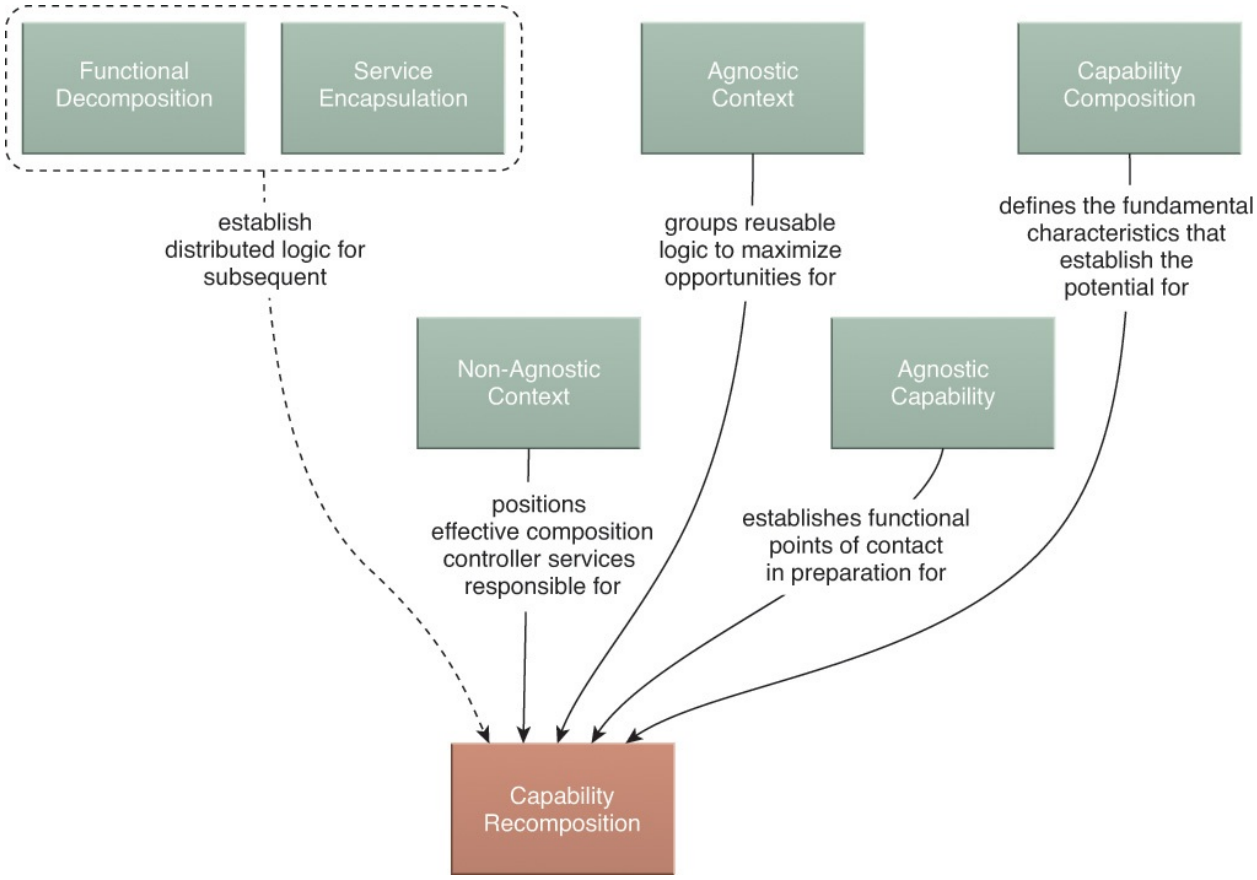


Figure 5.19 The repeated composability of services is core to service-orientation.

SOA Patterns

The steps explored in this chapter correspond to SOA patterns of the same names:

Functional Decomposition [344]

Service Encapsulation [359]

Agnostic Context [323]

Agnostic Capability [322]

Utility Abstraction [364]

Entity Abstraction [341]

Non-Agnostic Context [351]

Micro Task Abstraction [350]

Process Abstraction [353]

Capability Composition [328]

Capability Recomposition [329]

Combining these patterns into sequences can form the basis of primitive modeling processes.

Logic Centralization and Service Normalization

As more services are added to a service inventory, careful attention needs to be given to the respective service boundaries. This introduces the concept of service normalization. Service boundaries are defined on a functional basis and new logic introduced into a service inventory is first analyzed for its coherency in relation to the functional boundaries of existing services in order to avoid functional overlap. Functional overlap results in redundant logic, which can lead to increased maintenance overhead on an ongoing basis and when business requirements change. It can further lead to governance and configuration management issues, especially in cases where the redundant logic is owned by different groups within an organization.

The less functional overlap that is allowed in a service inventory, the less redundant logic exists, and the more normalized the service inventory becomes. Logic centralization is a technique that supports service normalization by centralizing logic in the form of single, normalized services (Figure 5.20).

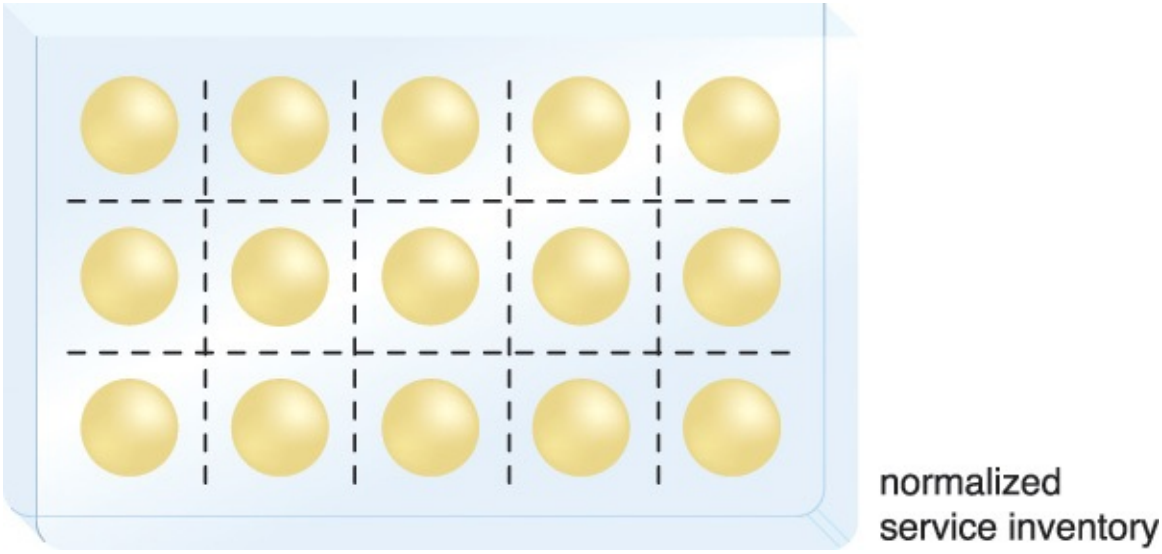


Figure 5.20 A service inventory comprised of services with published physical contracts. Each service has a distinct functional boundary, complementary to others and, ideally, without overlap.

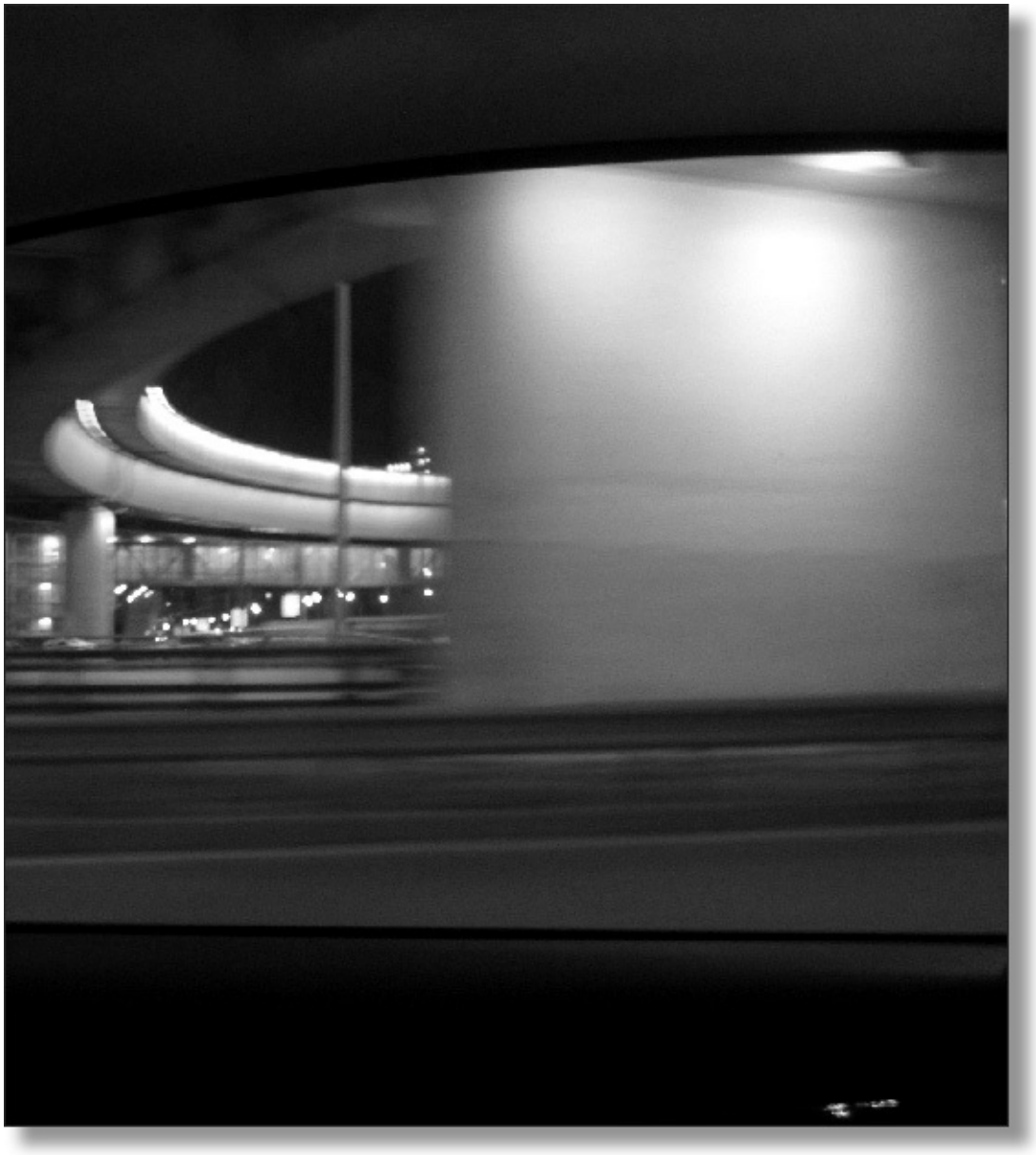
SOA Patterns

Service normalization and logic centralization are represented by the Service Normalization [361] and Logic Centralization [348] patterns, respectively.

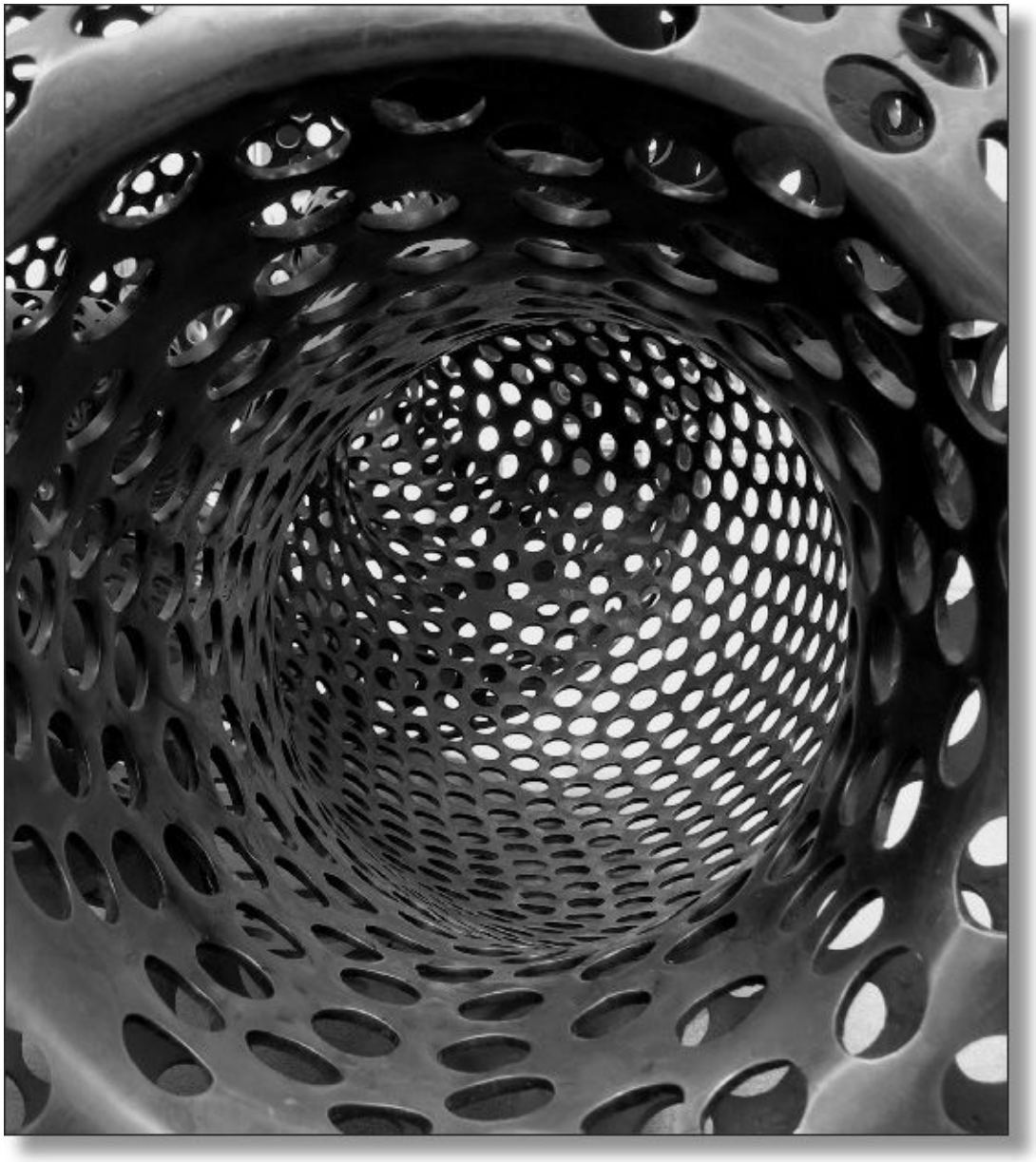
When applying Service Normalization [361] in support of Web services, the services are collectively modeled before their individual physical contracts (WSDL and XML Schema definitions) are created. This provides the opportunity for each Web service boundary to be planned out to ensure that it does not overlap with other services.

Because, within REST service implementations, the service contract is not “packaged” with the service architecture and logic, it is relatively easy for others in an IT department to add new REST services to a service inventory, particularly in the absence of a contract-first design approach. This tends to result in service capabilities with resource identifiers that perform functions redundant with those provided by existing REST services. Similarly, a new REST service may inadvertently add an entity service capability that belongs to the functional context of an existing REST entity service. This issue can also be addressed by applying Service Normalization [361]. Normalizing a REST-centric service inventory requires upfront analysis, established governance practices, and a “whole-of-inventory” perspective to be applied. Normalization makes it easier for service consumers to find and correctly use the functionality they need in a consistent, logically partitioned space of REST service capabilities grouped into distinct functional contexts.

Part II: Service-Oriented Analysis and Design



Chapter 6. Analysis and Modeling with Web Services and Microservices



This chapter provides a detailed step-by-step process for modeling Web service candidates.

6.1 Web Service Modeling Process

A service modeling process can essentially be viewed as an exercise in organizing the information we gathered in Steps 1 and 2 of the parent service-oriented analysis process that was described in Chapter 4. Figure 6.1 provides a generic service modeling process suitable for Web services that can be further customized. This chapter follows this generic service modeling process by describing each step and further providing case study examples.

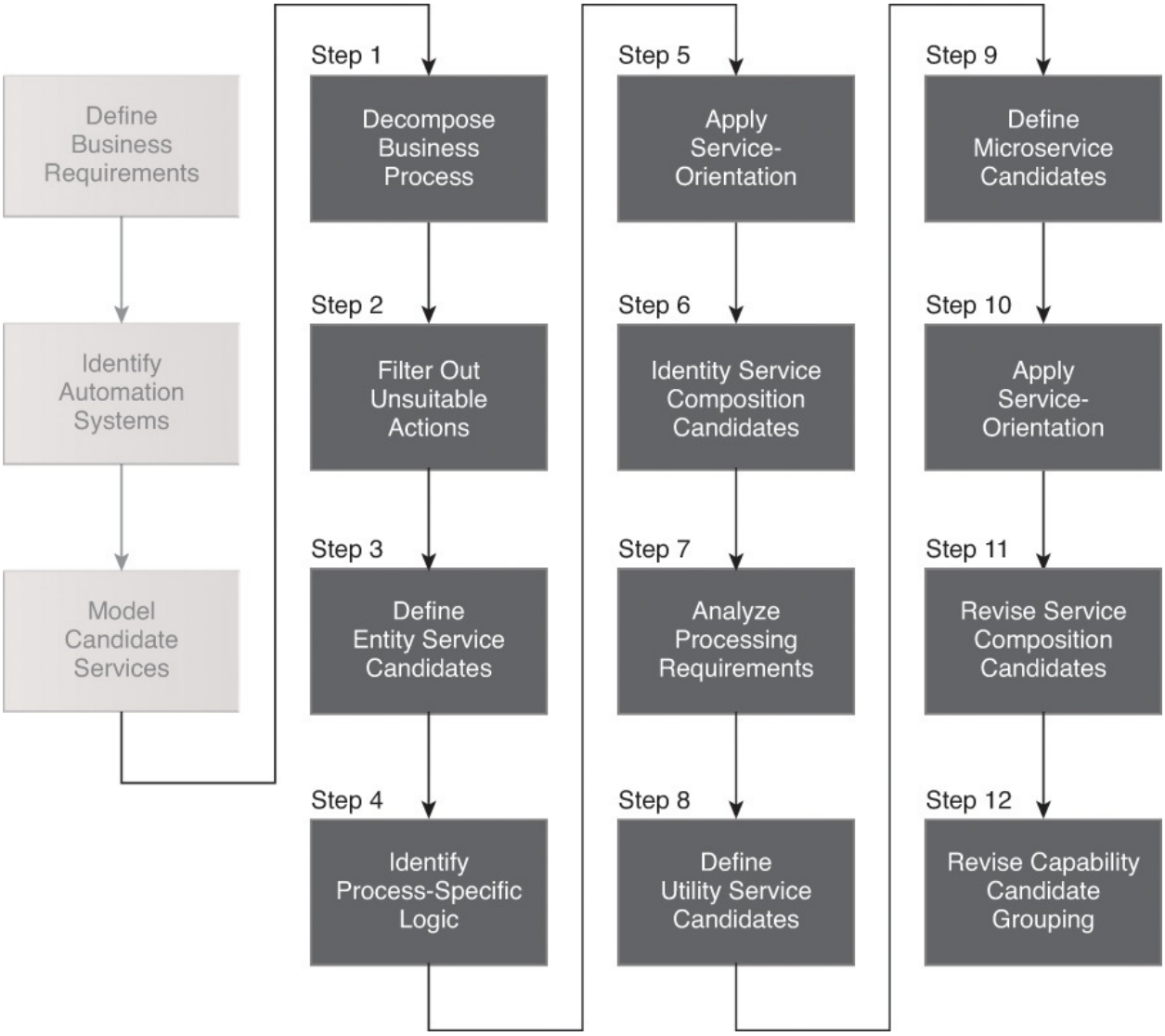


Figure 6.1 A sample service modeling process for Web services.

Case Study Example

TLS outsources a number of its employees on a contract basis to perform various types of specialized maintenance jobs. When these employees fill out their weekly timesheets, they are required to identify what portions of their time are spent at customer sites.

Currently, the amount of time for which a customer is billed is determined by an A/R clerk who manually enters hours from an appointment schedule that is published prior to the submission of timesheets.

Discrepancies arise when employee timesheet entries do not match the hours billed on customer invoices. To address this problem and streamline the overall process, TLS decides to integrate its third-party time tracking system with its large, distributed accounting solution.

The resulting Timesheet Submission business process is shown in Figure 6.2. Essentially, every timesheet that TLS receives from outsourced employees needs to undergo a series of verification steps. If the timesheet is verified successfully, the process ends and the timesheet is accepted. Any timesheet that fails verification is submitted to a separate rejection step prior to the process ending.

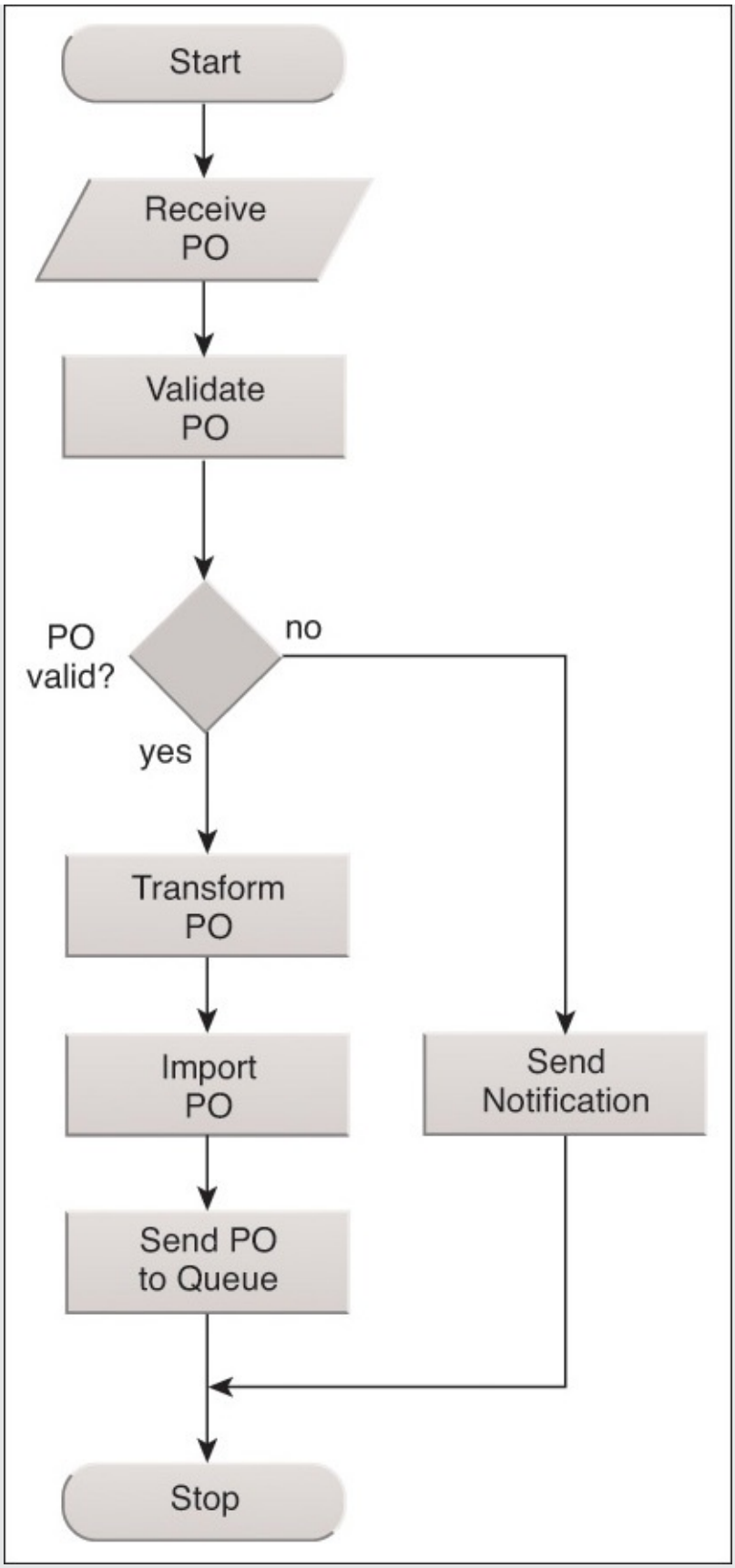

Start
Receive PO
Validate PO
PO valid?
no
yes
Transform PO
Import PO
Send Notification
Send PO to Queue
Stop

Figure 6.2 The TLS Timesheet Submission business process.

Step 1: Decompose the Business Process (into Granular Actions)

We begin by taking the documented business process and breaking it down into a series of granular process steps. The business process workflow logic needs to be decomposed into its most granular representation of processing steps, which may differ from the level of granularity at which the process steps were originally documented.

Case Study Example

Here is a breakdown of the current business process steps:

1. Receive Timesheet
2. Verify Timesheet
3. If Timesheet is Verified, Accept Timesheet Submission and End Process
4. Reject Timesheet Submission

Although it only consists of four steps at this point, there is more to this business process. The details are revealed as the TLS team decomposes the process logic. They begin with the *Receive Timesheet* step, which is split into two smaller steps: 1a. Receive Physical Timesheet Document

1b. Initiate Timesheet Submission

The *Verify Timesheet* step is actually a subprocess in its own right and can therefore be broken down into the following more granular steps: 2a. Compare Hours Recorded on Timesheet to Hours Billed to Clients

2b. Confirm That Authorization Was Given for Any Recorded Overtime Hours

2c. Confirm That Hours Recorded for Any Particular Project Do Not Exceed a Pre-Defined Limit for That Project 2d. Confirm That Total Hours Recorded for One Week Do Not Exceed a Pre-Defined Maximum for That Worker Upon subsequent analysis, TLS further discovers that the *Reject Timesheet Submission* process step can be decomposed into the following granular steps: 4a. Update the Worker’s Profile Record to Keep Track of

Rejected Timesheets 4b. Issue a Timesheet Rejection Notification Message to the Worker

4c. Issue a Notification to the Worker's Manager

Having drilled down the original process steps, TLS now has a larger amount of process steps. It organizes these steps into an expanded business process workflow (Figure 6.3): • Receive Timesheet

• Compare Hours Recorded on Timesheet to Hours Billed to Clients

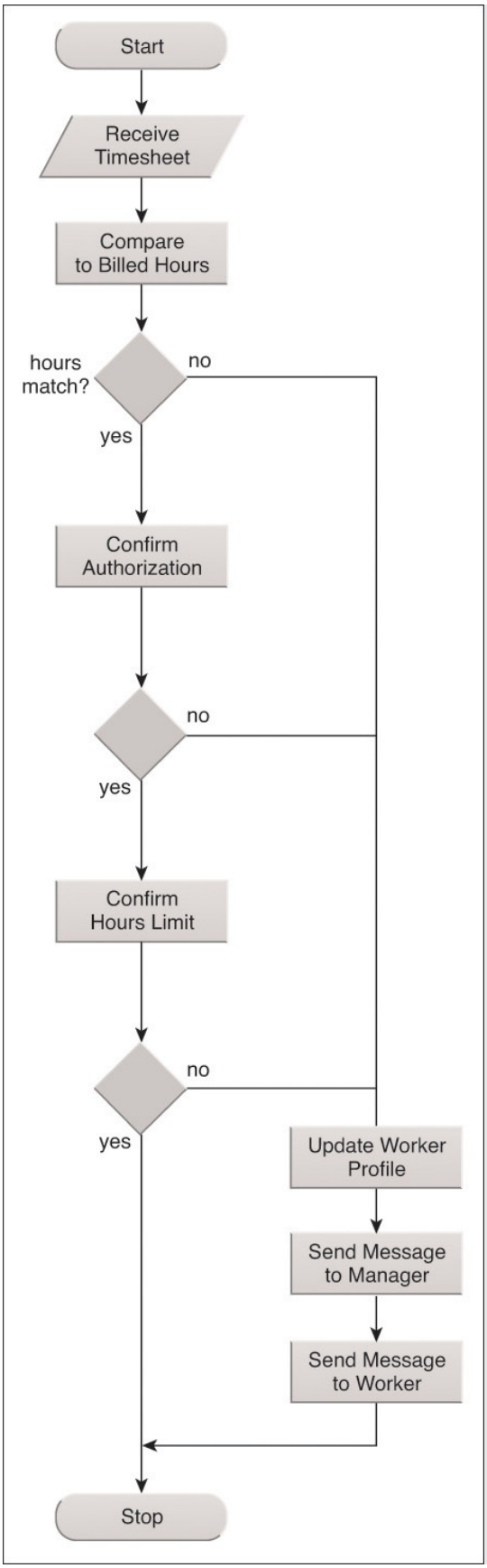

Start
Receive Timesheet
Compare to Billed Hours
hours match?
no
yes
Confirm Authorization
no
yes
Confirm Hours Limit
no
yes
Update Worker Profile
Send Message to Manager
Send Message to Worker
Stop

Figure 6.3 The revised TLS Timesheet Submission business process.

If Hours Do Not Match, Reject Timesheet Submission

- Confirm That Authorization Was Given for Any Recorded Overtime Hours
- If Authorization Confirmation Fails, Reject Timesheet Submission
- Confirm That Hours Recorded for Any Particular Project Do Not Exceed a Pre-Defined Limit for That Project • Confirm That Total Hours Recorded for One Week Do Not Exceed a Pre-Defined Maximum for That Worker • If Hours Recorded Confirmation Fails, Reject Timesheet Submission
- Reject Timesheet Submission
- Generate a Message Explaining the Reasons for the Rejection
- Issue a Timesheet Rejection Notification Message to the Worker
- Issue a Notification to the Worker's Manager
- If Timesheet Is Verified, Accept Timesheet Submission and End Process

Finally, TLS further simplifies the business process logic into the following set of granular actions: • Receive Timesheet

- Initiate Timesheet Submission
- Get Recorded Hours for Customer and Date Range
- Get Billed Hours for Customer and Date Range
- Compare Recorded Hours with Billed Hours
- If Hours Do Not Match, Reject Timesheet Submission
- Get Overtime Hours for Date Range
- Get Authorization
- Confirm Authorization
- If Authorization Confirmation Fails, Reject Timesheet Submission
- Get Weekly Hours Limit
- Compare Weekly Hours Limit with Recorded Hours
- If Hours Recorded Confirmation Fails, Reject Timesheet Submission

- Update Employee History
- Send Message to Employee
- Send Message to Manager
- If Timesheet Is Verified, Accept Timesheet Submission and End Process

Step 2: Filter Out Unsuitable Actions

Some steps within a business process can be easily identified as *not* belonging to the potential logic that should be encapsulated by a service candidate. These can include manual process steps that cannot or should not be automated and process steps performed by existing legacy logic for which service candidate encapsulation is not an option. By filtering out these parts, we are left with the processing steps most relevant to our service modeling process.

Case Study Example

After reviewing each of the business process steps, those that either cannot or do not belong in a service-oriented solution are removed. The following list revisits the decomposed actions. The first action is crossed out because it is performed manually by an accounting clerk.

- ~~Receive Timesheet~~
- Initiate Timesheet Submission
- Get Recorded Hours for Customer and Date Range
- Get Billed Hours for Customer and Date Range
- Compare Recorded Hours with Billed Hours
- If Hours Do Not Match, Reject Timesheet Submission
- Get Overtime Hours for Date Range
- Get Authorization
- Confirm Authorization
- If Authorization Confirmation Fails, Reject Timesheet Submission
- Get Weekly Hours Limit
- Compare Weekly Hours Limit with Recorded Hours
- If Hours Recorded Confirmation Fails, Reject Timesheet

Submission

- Update Employee History
- Send Message to Employee
- Send Message to Manager
- If Timesheet Is Verified, Accept Timesheet Submission and End Process

Each of the remaining actions is considered a service capability candidate.

Step 3: Define Entity Service Candidates

Review the processing steps that remain and determine one or more logical contexts with which these steps can be grouped. Each context represents a service candidate. The contexts you end up with will depend on the types of business services you have chosen to build. For example, task services will require a context specific to the process, whereas entity services will introduce the need to group processing steps according to their relation to previously defined entities. An SOA can also consist of a combination of these business service types.

It is important that you do not concern yourself with how many steps belong to each group. The primary purpose of this exercise is to establish the required set of contexts.

Equipping entity service candidates with additional capability candidates that facilitate future reuse is also encouraged. Therefore, the scope of this step can be expanded to include an analysis of additional service capability candidates not required by the current business process, but added to round out entity services with a complete set of reusable operations.

Case Study Example

TLS business analysts support the service modeling effort by producing an entity model relevant to the Timesheet Submission business process logic (Figure 6.4).

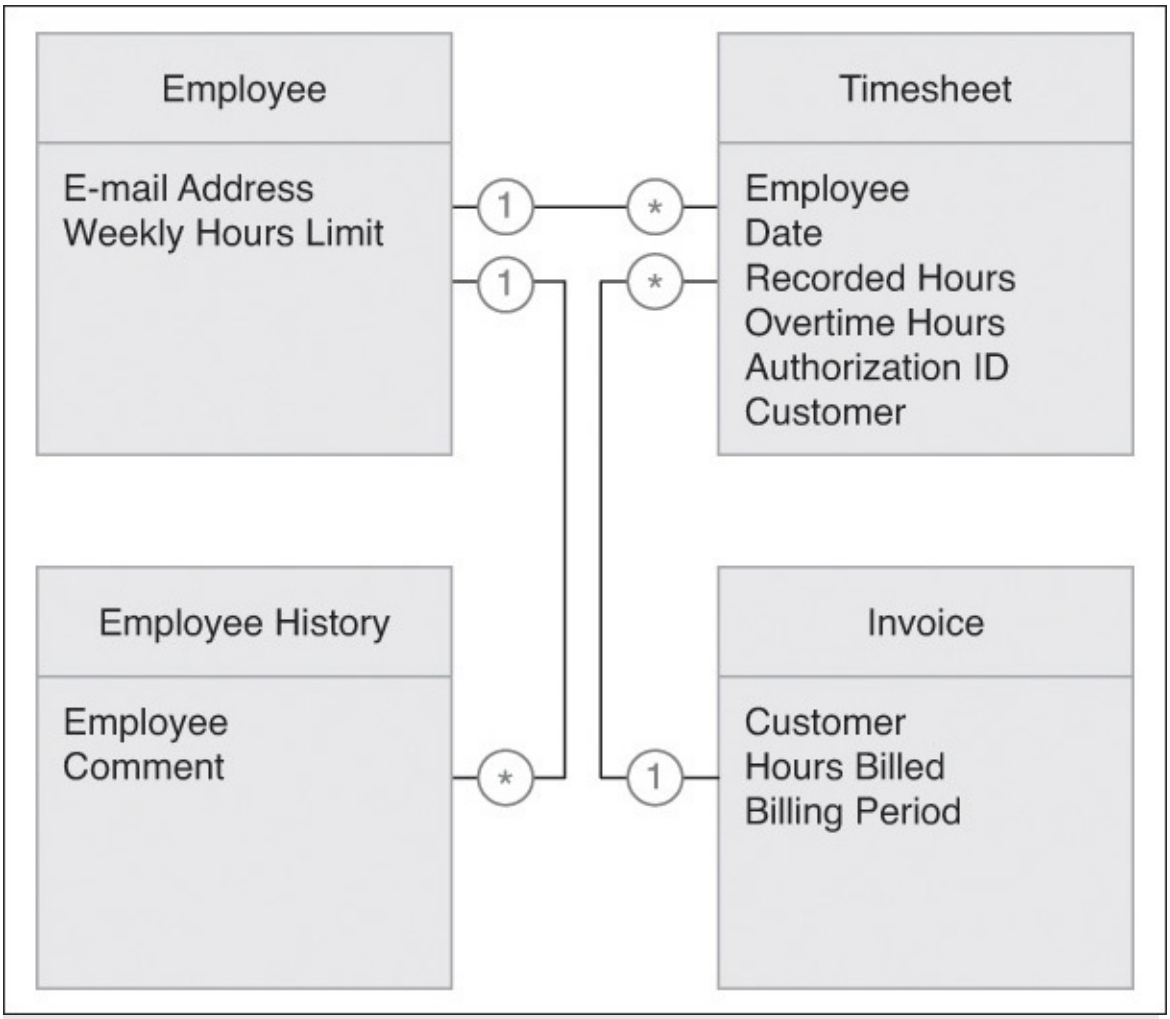


Figure 6.4 A TLS entity model displaying business entities pertinent to the Timesheet Submission business process.

The TLS team studies this model, along with the list of granular service capability candidates identified during the previous analysis step. They subsequently identify the service capability candidates considered agnostic. All those classified as non-agnostic are bolded, as follows: • **Initiate Timesheet Submission**

- Get Recorded Hours for Customer and Date Range
- Get Billed Hours for Customer and Date Range
- **Compare Recorded Hours with Billed Hours**
- **If Hours Do Not Match, Reject Timesheet Submission**
- Get Overtime Hours for Date Range
- Get Authorization
- **Confirm Authorization**

- **If Authorization Confirmation Fails, Reject Timesheet Submission**
- Get Weekly Hours Limit
- **Compare Weekly Hours Limit with Recorded Hours**
- **If Hours Recorded Confirmation Fails, Reject Timesheet Submission**
- Update Employee History
- Send Message to Employee
- Send Message to Manager
- **If Timesheet Is Verified, Accept Timesheet Submission and End Process**

First, the Timesheet entity is reviewed. It is decided that this entity warrants a corresponding entity service candidate simply called "Timesheet." Upon analysis of its attributes, TLS further determines that the following service capability candidates should be grouped with the entity service candidate: • Get Recorded Hours for Customer and Date Range

- Get Overtime Hours for Date Range
- Get Authorization

However, upon subsequent analysis, it is determined that the first two capability candidates could be made more reusable by removing the requirement that a date range be the only query criteria. Although this particular business process will always provide a date range, business analysts point out that other processes will want to request recorded or overtime hours based on other parameters. The result is a revised set of capability candidates, as shown in Figure 6.5.

Figure 6.5 The Timesheet service candidate.

Analysts then take a look at the Invoice entity. They again agree that this entity deserves representation as a standalone entity service candidate. They name this service “Invoice” and assign it the following capability candidate: • Get Billed Hours for Customer and Date Range

When the service-orientation principle of Service Reusability is again considered, the analysts decide to expand the scope of this service candidate by altering the function of the chosen capability candidate and then by adding a new one, as shown in Figure 6.6. Now service consumers can retrieve invoice-related customer information and billed hours information separately.

Figure 6.6 The Invoice service candidate.

The Employee and Employee History entities are reviewed next. Because they are closely related to each other, it is decided that they can be jointly represented by a single entity service candidate called "Employee." Two service capability candidates are assigned, resulting in the service candidate definition displayed in Figure 6.7.

Figure 6.7 The Employee service candidate.

The TLS team considers also adding a Send Notification service capability candidate to the Employee service candidate, but then determines that this functionality is best separated into a utility service candidate. As a result, the remaining two actions are put

aside for now until utility services are defined, later in this process:
• Send Message to Employee
• Send Message to Manager

Step 4: Identify Process-Specific Logic

Any parts of the business process logic remaining after we complete Step 3 will need to be classified as non-agnostic or specific to the business process. Common types of actions that fall into this category include business rules, conditional logic, exception logic, and the sequence logic used to execute the individual business process actions.

Note that not all non-agnostic actions necessarily become service capability candidates. Many process-specific actions represent decision logic and other forms of processing that are executed within the service logic.

Note

There may be sufficient information about the identified non-agnostic logic to determine whether any part of this logic may be suitable for encapsulation by one or more microservices. In this case, microservice candidates can be defined as part of this step together with task service candidates. However, it is recommended that you wait until Step 9 to formally define the necessary microservice(s) for this solution because upcoming service modeling steps can identify additional non-agnostic logic and can further assist with the definition of solution implementation and processing requirements.

Case Study Example

The following actions are considered non-agnostic because they are specific to the Timesheet Submission business process: • Initiate Timesheet Submission

- **Compare Recorded Hours with Billed Hours**
- **If Hours Do Not Match, Reject Timesheet Submission**
- **Confirm Authorization**
- **If Authorization Confirmation Fails, Reject Timesheet Submission**

- **Compare Weekly Hours Limit with Recorded Hours**
- **If Hours Recorded Confirmation Fails, Reject Timesheet Submission**
- **If Timesheet Is Verified, Accept Timesheet Submission and End Process**

The Initiate Timesheet Submission action forms the basis of a service capability candidate, as explained in the upcoming Timesheet Submission task service candidate description. The remaining actions are bolded to indicate that they represent logic that is carried out within the Timesheet Submission task service, upon execution of the Initiate Timesheet Submission action, which is renamed to the Start service capability candidate (Figure 6.8).

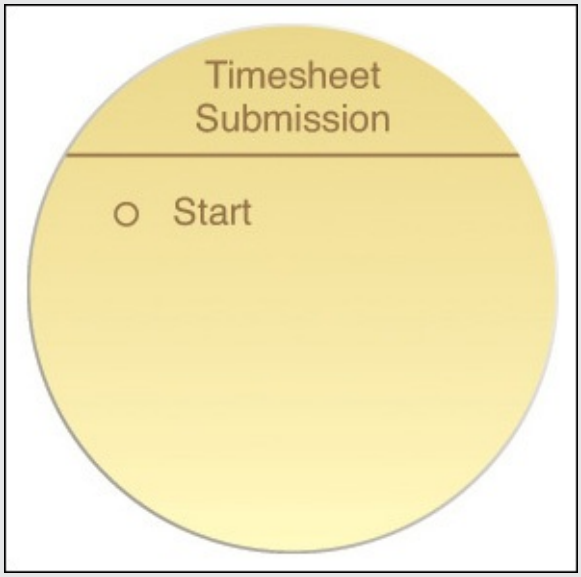


Figure 6.8 The Timesheet Submission service candidate with a single service capability that launches the automation of the Timesheet Submission business process.

Step 5: Apply Service-Orientation

This step gives us a chance to make adjustments and apply key service-orientation principles. Depending on the insight we may have as to the specific nature of logic that will be required within a given service candidate, we may have an opportunity to further augment the scope and structure of service candidates. Principles such as Service Loose Coupling (293), Service Abstraction (294), and Service Autonomy (297) may provide suitable considerations at this stage.

Note

The application of the Service Autonomy (297) principle in particular may raise considerations that could introduce the need for some of the identified logic to be encapsulated within microservices. In this case, microservice candidates can be defined as part of this step and will be subject to further review during Step 9, when microservices are formally defined.

Step 6: Identify Service Composition Candidates

Identify a set of the most common scenarios that can take place within the boundaries of the business process. For each scenario, follow the required processing steps as they exist now.

This exercise accomplishes the following:

- Provides insight as to how appropriate the grouping of your process steps is • Demonstrates the potential relationship between task and entity service layers • Identifies potential service compositions
- Highlights any missing workflow logic or processing steps

Ensure that, as part of your chosen scenarios, you include failure conditions that involve exception handling logic. Note also that any service layers you establish at this point are still preliminary and still subject to revisions during the design process.

Case Study Example

Figure 6.9 displays a preliminary service composition candidate comprised of task and entity service candidates. This composition model is the result of various composition scenarios mapped out by the TLS team to explore different success and failure conditions when carrying out the automation of the Timesheet Submission process.

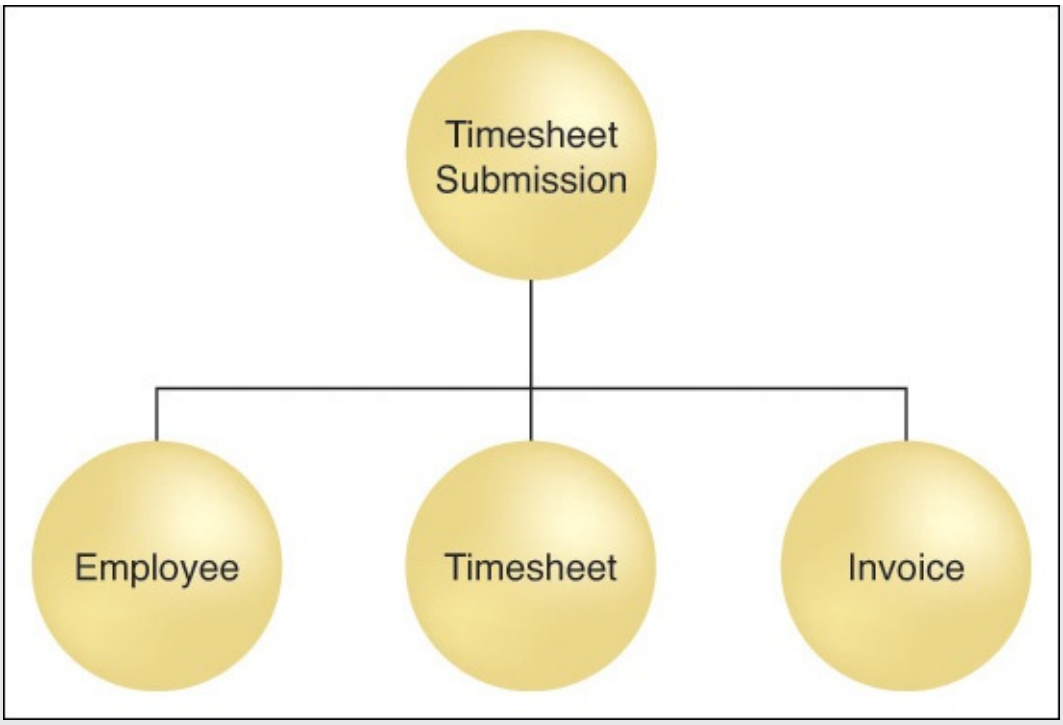


Figure 6.9 A look at the service composition candidate hierarchy that is formed as various service interaction scenarios are explored during this stage.

As a result of mapping different service activities within the boundaries of this service composition candidate, TLS feels confident that no further non-agnostic process logic is missing from what it has identified so far.

Step 7: Analyze Processing Requirements

By the end of Step 6, you will have created a business-centric view of your services layer. This view could very well include both utility and business service candidates, but the focus so far has been on representing business process logic.

This and the upcoming steps ask us to identify and dissect the underlying processing and implementation requirements of service candidates. We do this to abstract any further technology-centric service logic that may warrant the introduction of microservices or that may add to the utility service layer. To accomplish this, each processing step identified so far is required to undergo a mini-analysis.

Specifically, what we need to determine is:

- What underlying processing logic needs to be executed to process the action described by a given service capability candidate.
- Whether the required processing logic already exists or whether it needs to be newly developed.
- What resources external to the service boundary the processing logic may need to access—for example, shared databases, repositories, directories, legacy systems, *etc.*
- Whether any of the identified processing logic has specialized or critical performance and/or reliability requirements.
- Whether the identified processing logic has any specialized or critical implementation and/or environmental requirements.

Note that any information gathered during Step 2 of the parent service-oriented analysis process covered in Chapter 4 will be referenced at this point.

Case Study Example

Upon assessing the processing requirements for the identified service candidates and the overall business process logic, the TLS team can confirm that the Send Message to Employee and Send Message to Manager actions will need to be encapsulated as part of a utility service layer. Based on the information available about the known processing requirements and the eventual service implementation environment, they cannot identify any further utility-centric logic.

During the review of the non-agnostic process logic that is currently within the scope of the Timesheet Submission task service, architects realize that a discrepancy exists in processing requirements. In particular, the Confirm Authorization action encompasses logic that is required to access a proprietary clearance repository. This interaction has significantly greater SLA requirements than the rest of the non-agnostic process logic in relation to performance and failover.

Keeping this logic grouped with the other logic that is part of the Timesheet Submission task service could risk this logic not executing as per its required metrics. Therefore, it is suggested that it be separated into one or more microservice candidates that would eventually benefit from the type of highly autonomous implementation that could guarantee the required performance and

failover demands.

Step 8: Define Utility Service Candidates

In this step we break down each unit of agnostic processing logic into a series of granular actions. We need to be explicit about the labeling of these actions so that they reference the function they are performing. Ideally, we would not reference the business process step for which a given function is being identified.

Group these processing steps according to a pre-defined context. With utility service candidates, the primary context is a logical relationship between capability candidates. This relationship can be based on any number of factors, including: • Association with a specific legacy system

- Association with one or more solution components
- Logical grouping according to type of function

Various other issues are factored in after service candidates are subjected to the service-oriented design process. For now, this grouping establishes a preliminary utility service layer.

Case Study Example

Subsequent to assessing processing requirements for logic that may qualify for the utility service model, the TLS team revisits the Send Message to Employee and Send Message to Manager actions and groups them into a new reusable utility service, simply called Notification.

To make the service candidate more reusable, the two capability candidates are consolidated into one as shown in Figure 6.10.

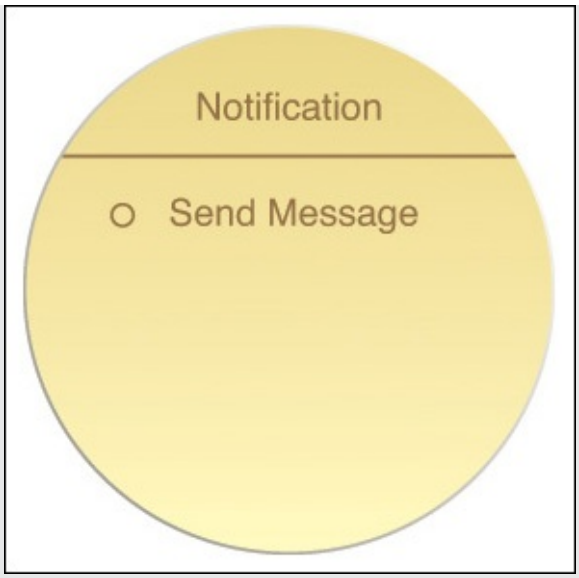


Figure 6.10 The Notification service candidate.

Note

Modeling utility service candidates is notoriously more difficult than entity service candidates. Unlike entity services where we base functional contexts and boundaries upon already-documented enterprise business models and specifications (such as taxonomies, ontologies, entity relationships, and so on), there are usually no such models for application logic. Therefore, it is common for the functional scope and context of utility service candidates to be continually revised during iterations of the service inventory analysis cycle.

Step 9: Define Microservice Candidates

We now turn our attention to the previously identified non-agnostic processing logic to determine whether any unit of this logic may qualify for encapsulation by a separate microservice. As discussed in Chapter 4, the microservice model can introduce a highly independent and autonomous service implementation architecture that can be suitable for units of logic with particular processing demands.

Typical considerations can include:

- Increased autonomy requirements
- Specific runtime performance requirements

- Specific runtime reliability or failover requirements
- Specific service versioning and deployment requirements

It is important to note that, due to their specialized implementation needs, the use of SOAP-based Web services may not be suitable for microservices, even when they are identified as part of a Web services-centric service modeling process. SOA architects are given the option to build microservices using alternative implementation technologies, which may introduce disparate or proprietary communication protocols.

SOA Patterns

The Dual Protocols [339] pattern provides a standardized manner of supporting primary and secondary communication protocols with the same service inventory.

Case Study Example

The Confirm Authorization action that is part of the Timesheet Submission task service candidate logic is separated to form the basis of the Confirm Authorization microservice candidate (Figure 6.11), a REST service that executes this logic via a Confirm capability candidate.

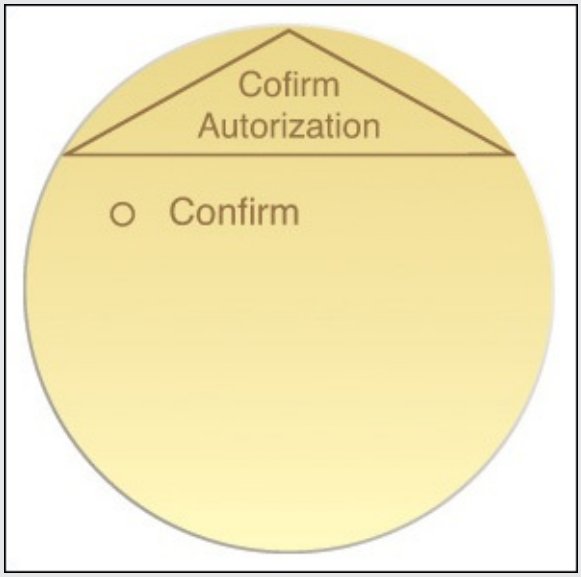


Figure 6.11 The Confirm Authorization service candidate.

For more information on service modeling steps distinct to REST services, see Chapter 7.

Step 10: Apply Service-Orientation

This step is a repeat of Step 7, provided here specifically for any new utility service candidates that may have emerged from the completion of Steps 8 and 9.

Step 11: Revise Service Composition Candidates

Revisit the original scenarios you identified in Step 6 and run through them again, this time incorporating the new utility service and capability candidates as well. This will result in the mapping of elaborate activities that bring expanded service compositions to life. Be sure to keep track of how business service candidates map to underlying utility service candidates during this exercise.

Case Study Example

With the introduction of the Notification utility service and the Verify Timesheet microservice, the complexion of the Timesheet Submission composition hierarchy changes noticeably, as illustrated in Figure 6.12.

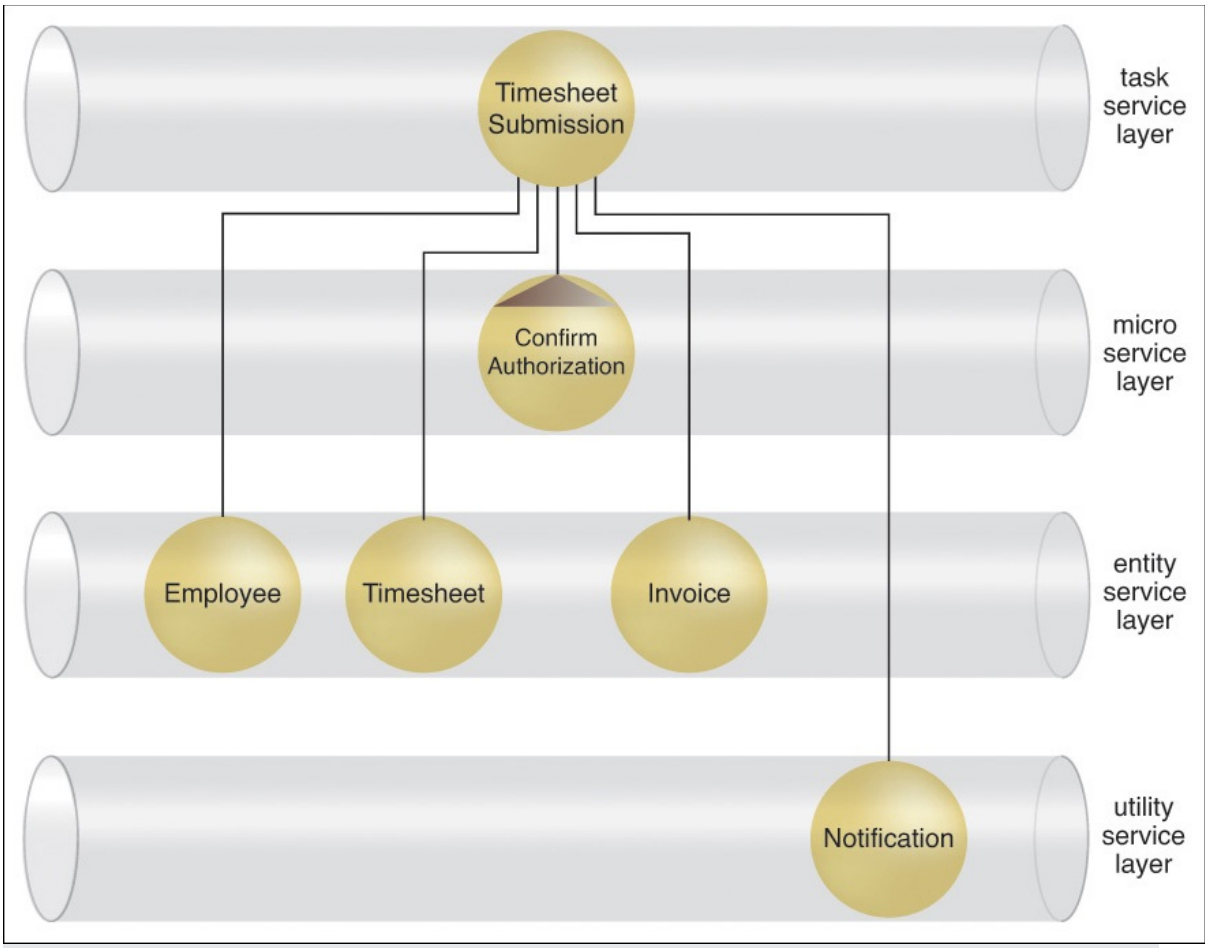


Figure 6.12 The revised service composition candidate incorporating the new utility service and microservice.

Step 12: Revise Capability Candidate Grouping

Performing the mapping of the activity scenarios from Step 11 will usually result in changes to the grouping and definition of service capability candidates. It may also highlight any omissions in any further required processing steps, resulting in the addition of new service capability candidates and possibly even new service candidates.

Note

This process description assumes that this is the first iteration through the service modeling process. During subsequent iterations, additional steps need to be incorporated to check for the existence of relevant service candidates and service capability candidates.

Chapter 7. Analysis and Modeling with REST Services and Microservices

This chapter provides a detailed step-by-step process for modeling REST service candidates.

7.1 REST Service Modeling Process

The incorporation of resources and uniform contract features adds new dimensions to service modeling. When we are aware that a given service candidate is being modeled specifically for a REST implementation, we can take these considerations into account by extending the service modeling process to include steps to better shape the service candidate as a basis for a REST service contract.

The REST service modeling process shown in Figure 7.1 provides a generic set of steps and considerations tailored for modeling REST services. This chapter describes each process step and is further supplemented with case study examples.

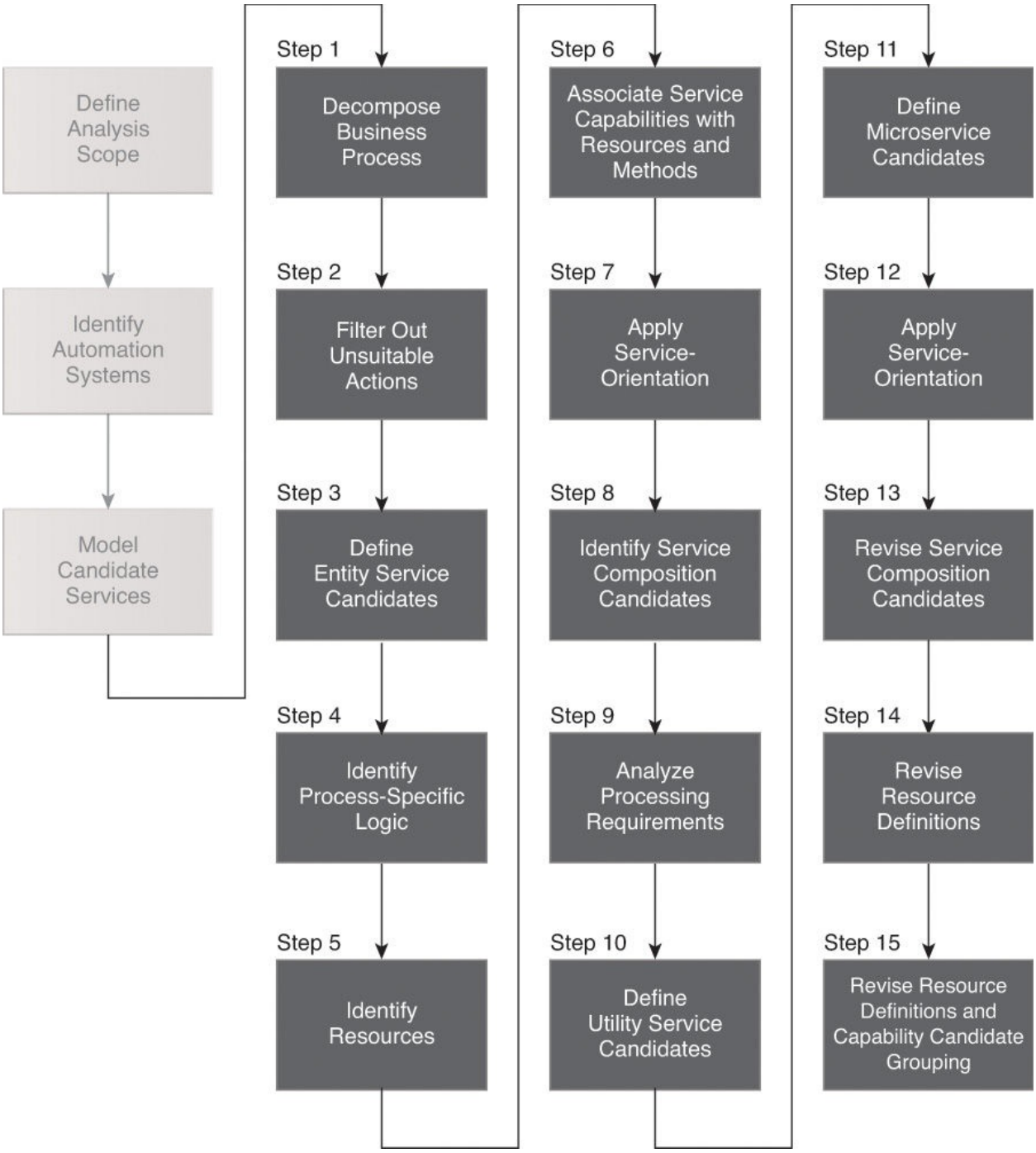


Figure 7.1 A sample service modeling process for REST services.

Case Study Example

MUA architects are dedicated to adopting SOA and applying service-orientation as part of a key strategy to consolidate systems and data. They decide to focus on entity services that track the information assets of the various campuses. This initial set of

services is to be deployed on the main campus first, so that IT staff can monitor maintenance requirements. Individual campuses are then to build solutions based on the same centralized service inventory. Solutions that introduce new task services will be allocated to virtual machines in the main campus to allow them to be moved to independent hardware and onto dedicated server farms, if the need arises in the future.

Existing MUA charter agreements with partner schools explicitly refer to the need to acknowledge individual academic achievements. This makes the correct conferral of awards important to the reputation of MUA and its elite students.

MUA assembles a service modeling team comprised of SOA architects, SOA analysts, and business analysts. The team begins with a REST service modeling process for the Student Achievement Award Conferral business process. As detailed in Figure 7.2, this business process logic represents the procedures followed for the assessment, conference, and rejection of individual achievement award applications submitted by students. An application that is approved results in the conferral of the achievement award and a notification of the conferral to the student. An application that is rejected results in a notification of the rejection to the student.

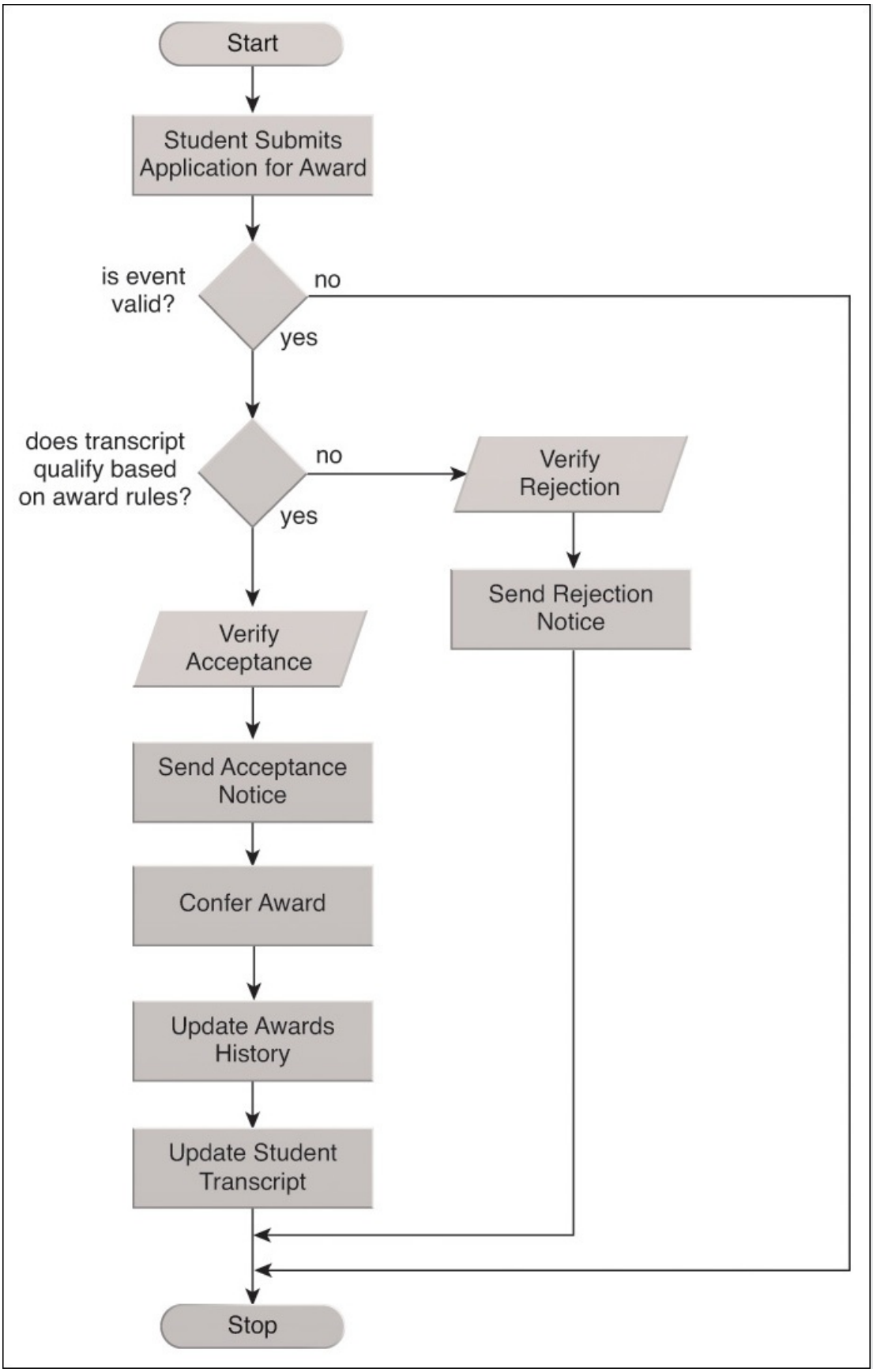

Start
Student Submits Application for Award
is event valid?
no
yes
does transcript qualify based on award rules?
no
yes
Verify Rejection
Send Rejection Notice
Verify Acceptance
Send Acceptance Notice
Confer Award
Update Awards History
Update Student Transcript
Stop

Figure 7.2 The Student Award Conferral business process.

Step 1: Decompose Business Process (into Granular Actions)

Let's take the documented business process and break it down into a series of granular process steps. This requires further analysis of the process logic, during which we attempt to decompose the business process into a set of individual granular actions.

Case Study Example

The original Student Award Conferral business process is broken down into the following granular actions:

- Initiate Conferral Application
- Get Event Details
- Verify Event Details
- If Event is Invalid or Ineligible for Award, End Process
- Get Award Details
- Get Student Transcript
- Verify Student Transcript Qualifies for Award Based on Award Conferral Rules
- If Student Transcript Does Not Qualify, Initiate Rejection
- Manually Verify Rejection
- Send Rejection Notice
- Manually Verify Acceptance
- Send Acceptance Notice
- Confer Award
- Record Award Conferral in Student Transcript
- Record Award Conferral in Awards Database
- Print Hard Copy of Award Conferral Record
- File Hard Copy of Award Conferral Record

Step 2: Filter Out Unsuitable Actions

Not all business process logic is suitable for automation and/or encapsulation by

a service. This step requires us to single out any of the granular actions identified in Step 1 that do not appear to be suitable for subsequent REST service modeling steps. Examples include manual process steps that need to be performed by humans and business automation logic being carried out by legacy systems that cannot be wrapped by a service.

Case Study Example

After assessing each of the decomposed actions, a subset is identified as being unsuitable for automation or unsuitable for service encapsulation, as indicated by the crossed-out items.

- Initiate Conferral Application
- Get Event Details
- Verify Event Details
- If Event is Invalid or Ineligible for Award, End Process
- Get Award Details
- Get Student Transcript
- Verify Student Transcript Qualifies for Award Based on Award Conferral Rules
- If Student Transcript Does Not Qualify, Initiate Rejection
- ~~Manually Verify Rejection~~
- Send Rejection Notice
- ~~Manually Verify Acceptance~~
- Send Acceptance Notice
- ~~Confer Award~~
- Record Award Conferral in Student Transcript
- Record Award Conferral in Awards Database
- Print Hard Copy of Award Conferral Record
- ~~File Hard Copy of Award Conferral Record~~

Step 3: Define Entity Service Candidates

By filtering out unsuitable actions during Step 2, we are left with only those actions relevant to our REST service modeling effort.

A primary objective of service-orientation is to carry out a separation of

concerns whereby agnostic logic is cleanly partitioned from non-agnostic logic. By reviewing the actions that have been identified so far, we can begin to further separate those that have an evident level of reuse potential. This essentially provides us with a preliminary set of agnostic service capability candidates.

We then determine how these service capability candidates should be grouped to form the basis of functional service boundaries.

Common factors we can take into account include:

- Which service capability candidates defined so far are closely related to each other?
- Are identified service capability candidates business-centric or utility-centric?
- What types of functional service contexts are suitable, given the overarching business context of the service inventory?

The first consideration on the list requires us to group capability candidates based on common functional contexts. The second item pertains to the organization of service candidates within logical service layers based on service models. Due to the business-centric level of documentation that typically goes into the authoring of business process models and specifications and associated workflows, the emphasis during this step will naturally be more on the definition of entity service candidates. The upcoming Define Utility Service Candidates step is dedicated to developing the utility service layer.

The third item on the preceding list of factors relates to how we may choose to establish functional service boundaries not only in relation to the current business process we are decomposing, but also in relation to the overall nature of the service inventory. This broader consideration helps us determine whether there are generic functional contexts we can define that will be useful for the automation of multiple business processes.

SOA Patterns

Both the previously referenced Logic Centralization [348] and Service Normalization [361] patterns play a key role during this step to ensure we keep agnostic service candidates aligned to each other, without allowing functional overlap.

Case Study Example

By analyzing the remaining actions from Step 2, the MUA service

modeling team identifies and categorizes those actions considered agnostic. Those that are classified as non-agnostic are in bold:

- **Initiate Conferral Application**
- Get Event Details
- **Verify Event Details**
- **If Event is Invalid or Ineligible for Award, Cancel Process**
- Get Award Details
- Get Student Transcript
- **Verify Student Transcript Qualifies for Award Based on Award Conferral Rules**
- **If Student Transcript Does Not Qualify, Initiate Rejection**
- Send Rejection Notice
- Send Acceptance Notice
- Record Award Conferral in Student Transcript
- Record Award Conferral in Awards Database
- Print Hard Copy of Award Conferral Record

Agnostic actions are classified as preliminary service capability candidates and are grouped accordingly into service candidates, as follows.

Event Service Candidate

The original Get Event Details action is positioned as a Get Details service capability candidate as part of an entity service candidate named Event (Figure 7.3).

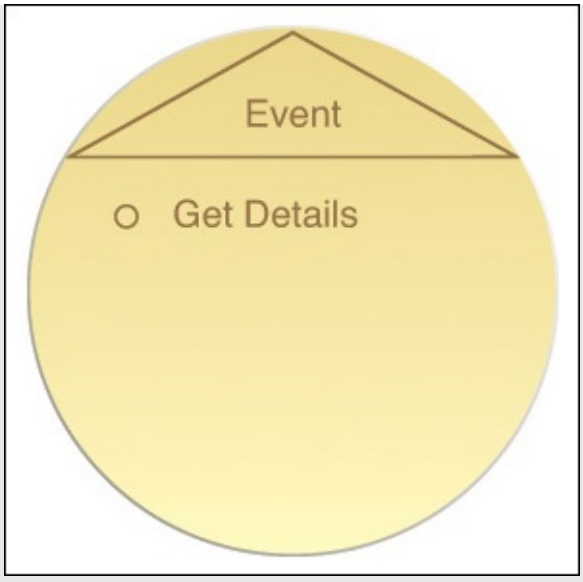


Figure 7.3 The Event service candidate with one service capability candidate.

Note that it was determined that the Verify Event Details action was not agnostic because it carried out logic specific to the Student Award Conferral process.

Award Service Candidate

As a central part of this business process, the Award business entity becomes the basis of an Award entity service candidate (Figure 7.4).

Figure 7.4 The Award service candidate with three service capability candidates, including two that are based on the same action.

The Get Award Details action establishes a Get Details service

capability candidate. The Record Award Conferral in Awards Database action is split into two service capability candidates:

- Confer
- Update History

The Confer capability is required to officially issue an award for an event, which requires updates in the internal MUA Awards database, as well as an update to an external National Academic Recognition System shared by schools throughout the U.S.

Furthermore, based on the award conferral policies, this service capability is required to issue a conferral notification and forward the award conferral record information to be printed in hard copy format. This relates to the following three actions:

- Send Rejection Notice
- Send Acceptance Notice
- Print Hard Copy of Award Conferral Record

The MUA team considers including this logic within the Award entity service, but then decides that the Confer service capability will instead invoke corresponding utility services to perform these functions automatically, upon each conferral.

The Update History capability will issue a further update of student and event details within a separate part of the internal Awards database. It is deemed necessary to keep the capabilities separate because the Update History capability can be used independently and for different purposes than the Confer capability.

Student Service Candidate

The need for a Student entity service within a school is self-evident. This service will eventually provide a wide range of student-related functions. In support of the Student Award Conferral business process specifically, the Get Student Transcript and Record Award Conferral in Student Transcript actions are positioned as individual service capability candidates named Get Transcript and Update Transcript (Figure 7.5).

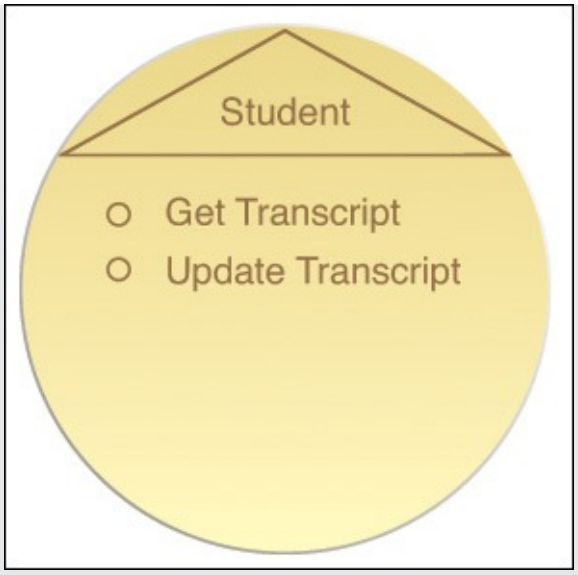


Figure 7.5 The Student service candidate with two service capability candidates.

As previously mentioned, the following three remaining actions are put aside for when utility services are modeled, later in this process:

- Send Rejection Notice
- Send Acceptance Notice
- Print Hard Copy of Award Conferral Record

Step 4: Identify Process-Specific Logic

Process-specific logic is separated into its own logical service layer. For a given business process, this type of logic is commonly grouped into a task service or a service consumer acting as the composition controller.

Case Study Example

The following actions are considered non-agnostic because they are specific to the Student Award Conferral business process:

- Initiate Conferral Application
- **Verify Event Details**
- **If Event is Invalid or Ineligible for Award, End Process**
- **Verify Student Transcript Qualifies for Award Based on Award Conferral Rules**
- **If Student Transcript Does Not Qualify, Initiate Rejection**

The first action on this list forms the basis of a service capability candidate, as explained shortly in the Confer Student Award task service candidate description. The remaining actions in bold do not correspond to service capability candidates. Instead, they are identified as logic that occurs internally within the Confer Student Award task service.

Confer Student Award Service Candidate

The Initiate Conferral Application action is translated into a simple Start service capability candidate as part of a Confer Student Award task service candidate (Figure 7.6). It is expected that the Start capability will be invoked by a separate software program, which would be acting as a composition initiator.

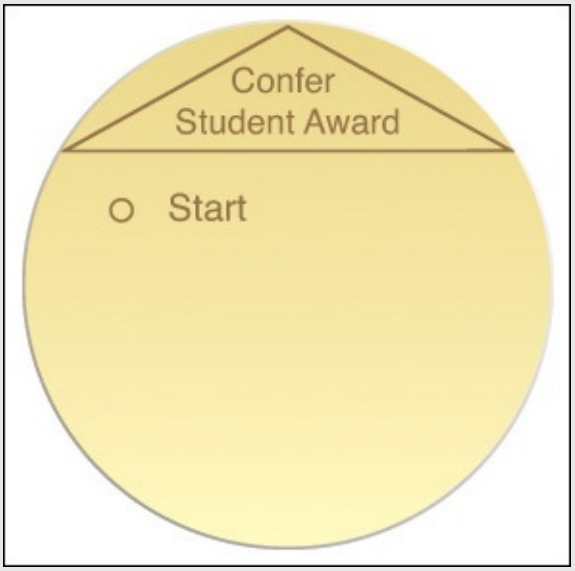


Figure 7.6 The Confer Student Award task service candidate with a single service capability that launches the automation of the Student Award Conferral business process.

Step 5: Identify Resources

By examining the functional contexts associated with individual actions, we can begin to make a list of how these contexts relate to or form the basis of resources. It can be helpful to further qualify identified resources as agnostic (multipurpose) or non-agnostic (single-purpose), depending on how specific we determine their usage and existence are to the parent business process.

Step 3 explained how labeling a service candidate or a service capability candidate as “agnostic” has significant implications as to how we approach the

modeling of that service. This is not the case with resources. From a modeling perspective, agnostic resources can be incorporated into agnostic service and capability candidates without limitation. The benefit to identifying agnostic resources is to earmark them as parts of the enterprise that are likely to be shared and reused more frequently than non-agnostic resources. This can help us prepare necessary infrastructure or perhaps even limit their access in how we model (and subsequently design) the service capabilities that encompass them.

Note that resources identified at this stage can be expressed using the forward slash as a delimiter. This is not intended to result in URL-compliant statements; rather, it is a means by which to recognize the parts of service capability candidates that pertain to resources. Similarly, modeled resources are intentionally represented in a simplified form. Later, in the service-oriented design stage, the syntactically correct resource identifier statements are used to represent resources, including any necessary partitioning into multi-part URL statements (as per resource identifier syntax standards being used).

Case Study Example

Subsequent to a review of the processing requirements of the service capability candidates defined so far, the following potential resources are identified:

- *Process*
- *Application*
- *Event*
- *Award*
- *Student Transcript*
- ***Notice Sender***
- ***Printer***

Before proceeding, the MUA service modeling team decides to further qualify the *Process* and *Application* resource candidates to better associate them with the nature of the overarching business processing logic, as follows:

- *Student Award Conferral Process*
- *Conferral Application*

These qualifiers help distinguish similar resources that may exist as other forms of applications or rules.

Because the service modeling process has, so far, already produced a set of entity services, each of which represents a business entity, it is further decided to establish some preliminary mapping between identified resources and entities, as shown in Table 7.1.

Entity	Resource
Event	/Event/
Award	/Award/
Student	/Student Transcript/

Table 7.1 Mapping business entities to resources.

The bolded resources in the preceding list are put aside for when utility services will be modeled, later in this process. Additional resources are not mapped because they do not currently relate to known business entities. They may end up being mapped during future iterations of the service modeling process.

Step 6: Associate Service Capabilities with Resources and Methods

We now associate the service capability candidates defined in Steps 3 and 4 with the resources defined in Step 5, as well as with available uniform contract methods that may have been established. If we discover that a given service capability candidate requires a method that does not yet exist in the uniform contract definition, the method can be proposed as input for the next iteration of the Model Uniform Contract task that is part of the service inventory analysis cycle.

We continue to use the same service candidate and service capability candidate notation, but we append service capability candidates with their associated method plus resource combinations. This allows for a descriptive and flexible expression of a preliminary service contract that can be further changed and refined during subsequent iterations of the service-oriented analysis process.

Note

At this stage it is common to associate actions with regular HTTP methods, as defined via uniform contract modeling efforts. Complex

methods can be comprised of pre-defined sets and/or sequences of regular method invocations. If complex methods are defined at the service modeling stage, then they can also be associated as appropriate.

Case Study Example

The MUA service modeling team continues to expand upon their original service candidate definitions by adding the appropriate uniform contract methods and resources, as follows.

Confer Student Award Service Candidate (Task)

The business document required as the primary input to kick off the Student Award Conferral business process is the application submitted by the student. It was initially assumed that an *Application* resource would be required to represent this document. However, upon further analysis, it turns out that all the Start service capability candidate needs is a POST method to forward the application document to a resource named after the business process itself (Figure 7.7).

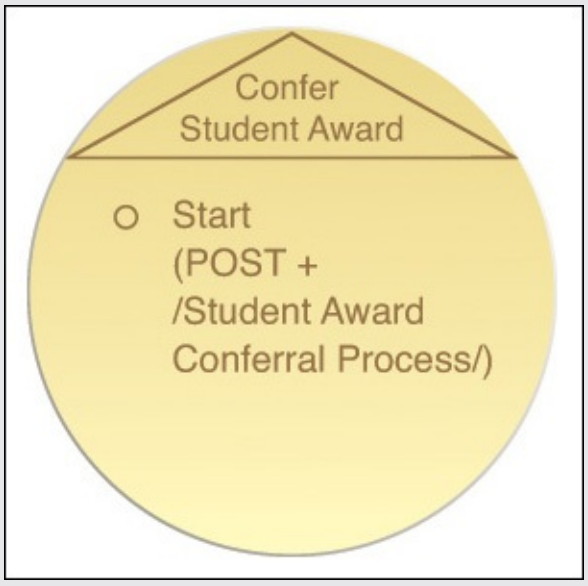


Figure 7.7 The Confer Student Award service candidate with method and resource association.

Event Service Candidate (Entity)

The sole Get Details service capability candidate is appended with

the GET method and the *Event* resource (Figure 7.8).

Figure 7.8 The Event service candidate with method and resource association.

Award Service Candidate (Entity)

The Get Details service capability is correspondingly associated with a GET method plus *Award* resource combination. The Confer and Update History service capability candidates each require input data that will update resource data, and therefore are expanded with a preliminary POST method and the *Awards* resource (Figure 7.9). This method may later be refined during the service-oriented design phase.

Figure 7.9 The Award service candidate with method and resource associations.

Student Service Candidate (Entity)

The Get Transcript service capability candidate is associated with the GET method and the *Student Transcript* resource. The Update Transcript is appended with the POST method together again with the *Student Transcript* resource (Figure 7.10).

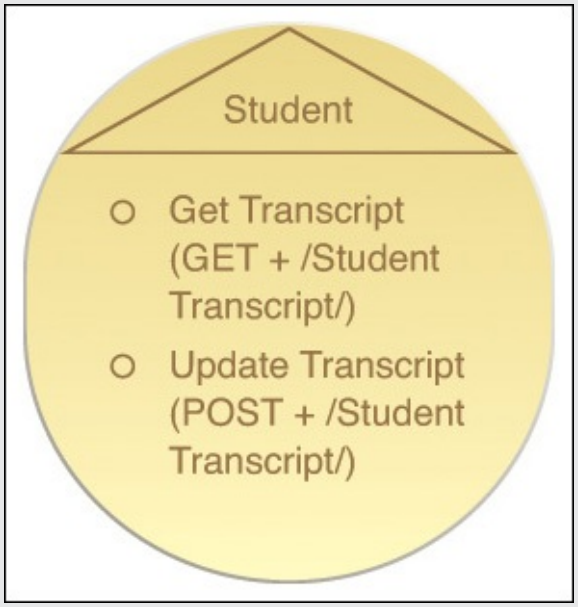


Figure 7.10 The Student service candidate with method and resource associations.

Step 7: Apply Service-Orientation

The business process documentation we used as input for the service modeling process may provide us with a level of knowledge as to the underlying processing required by each of the identified REST service capability candidates. Based on this knowledge, we may be able to further shape the definition and scope of service capabilities, as well as their parent service candidates, by taking a relevant subset of the service-orientation principles into consideration.

Case Study Example

When applying this step, the MUA service modeling team is faced with various practical concerns, based on what participating SOA architects can provide in terms of knowledge of the implementation environment that the services will be deployed in.

For example, they identify that a given set of resources is related to data provided by a large legacy system. This impacts functional service boundaries by the extent to which the Service Autonomy (297) principle can be applied.

Step 8: Identify Service Composition Candidates

Here we document the most common service capability interactions that can take place during the execution of the business process logic. Different interactions are mapped out based on the success and failure scenarios that can occur during the possible action sequences within the business process workflow.

Mapping these interaction scenarios to the required service capability candidates enables us to model candidate service compositions. It is through this type of view that we can get a preview of the size and complexity of potential service compositions that result from how we defined the scope and granularity of agnostic and non-agnostic service candidates (and capability candidates) so far. For example, if we determine that the service composition will need to involve too many service capability invocations, we still have an opportunity to revisit our service candidates.

It is also at this stage that we begin to take a closer look at data exchange requirements (because for services to compose each other, they must exchange data). This may provide us with enough information to begin identifying required media types based on what has already been defined for the uniform

contract. Alternatively, we may determine the need for new media types that have not yet been modeled. In the latter case, we may be gathering information that will act as input for the Model Uniform Contract task that is part of the service inventory analysis cycle (as explained later in the *Uniform Contract Modeling and REST Service Inventory Modeling* section).

Note

The depth of service compositions can particularly impact method definition. It is important to pose questions about the possible failure scenarios that can occur during service composition execution.

Case Study Example

The MUA service modeling team explores a set of service composition scenarios that correspond to success and failure conditions that may arise when the Student Award Conferral process is executed.

Figure 7.11 illustrates the composition hierarchy of service candidates that is relatively consistent across these scenarios. In each case, the Confer Student Award task service invokes the Event, Award, and Student entity services. The Award entity service further composes the Notification utility service to issue acceptance or rejection notifications and, if the award is conferred, the Document utility service to print the award record.

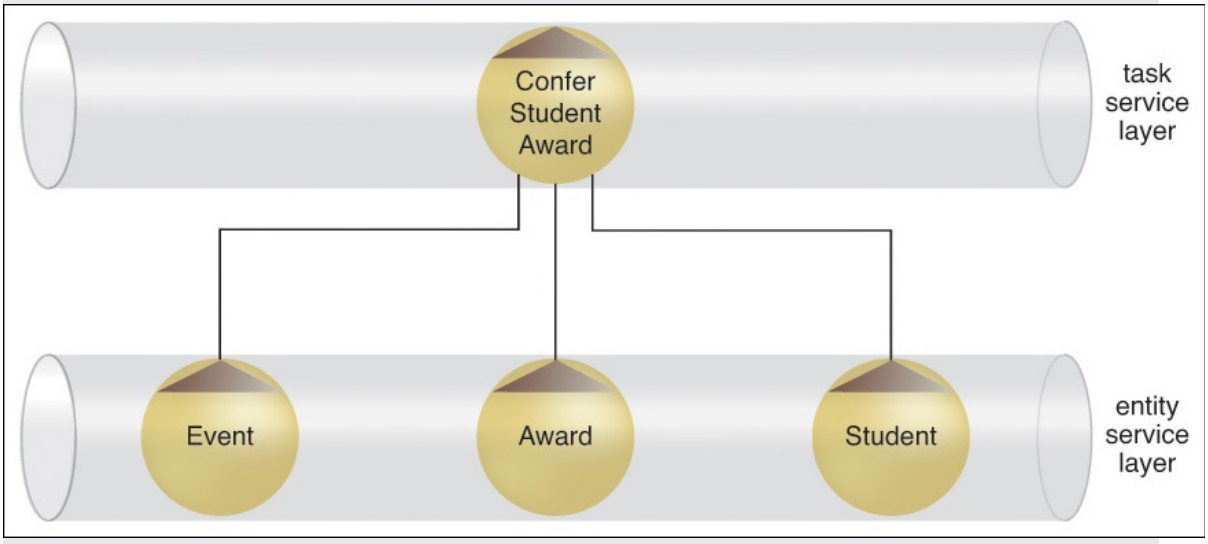


Figure 7.11 A look at the service composition candidate hierarchy that is formed as various service interaction scenarios are explored during this stage.

Note

This next series of steps is optional and more suited for complex business processes and larger service inventory architectures. It requires that we more closely study the underlying processing requirements of all service capability candidates in order to abstract further utility service candidates.

Step 9: Analyze Processing Requirements

As mentioned in the description for Step 3, the emphasis so far in this service modeling process will likely have been on business-centric processing logic. This is to be expected when working with business process definitions that are primarily based on a business view of automation. However, it is prudent to look under the hood of the business logic defined so far in order to identify the need for any further application logic.

To accomplish this, we need to consider the following:

- Which of the resources identified so far can be considered utility-centric?
- Can actions performed on business-centric resources be considered utility-centric (such as reporting actions)?
- What underlying application logic needs to be executed in order to process the actions and/or resources encompassed by a service capability candidate?
- Does any required application logic already exist?
- Does any required application logic span application boundaries? (In other words, is more than one system required to complete the action?)

Note that information gathered during the Identify Automation Systems step of the parent service-oriented analysis process will be referenced at this point.

Case Study Example

The MUA team carefully studies the processing requirements of the logic that will need to be encapsulated by the service candidates defined so far. They confirm that, beyond the already-identified

Send Rejection Notice, Send Acceptance Notice, and Print Hard Copy of Award Conferral Record actions, there appear to be no further utility-centric functions required. This then sets the stage for the upcoming Define Utility Services (and Associate Resources and Methods) step during which these actions, together with the previously identified utility-centric resources, will act as the primary input for utility service candidate definition.

However, while no new utility-centric processing requirements were identified, a concern was raised specifically regarding the non-agnostic Verify Student Transcript Qualifies for Award Based on Award Conferral Rules action that is currently encapsulated as part of the Confer Student Award task service. Architects discover that to complete this action, an external Rules utility service will need to be composed and invoked to complete the verification. Infrastructure statistics show that this existing Rules service is widely used and frequently reaches its usage thresholds, resulting in response delays and, during peak usage periods, occasional response rejections.

This raises concerns by business analysts who point out that there are policy-driven requirements that need to be fulfilled by carrying out an immediate verification of student transcripts. Further, and more importantly, after a verification has occurred, it is legally binding and cannot be reversed.

As a result, the MUA team classifies the Verify Student Transcript Qualifies for Award Based on Award Conferral Rules action as having critical and specialized processing requirements that cannot be met if it were to remain as part of the task service implementation. They therefore determine that this logic needs to be moved to a dedicated microservice.

Step 10: Define Utility Service Candidates (and Associate Resources and Methods)

In this step we group utility-centric processing steps according to pre-defined contexts. With utility service candidates, the primary context is a logical relationship between capability candidates. This relationship can be based on any number of factors, including:

- Association with a specific legacy system
- Association with one or more solution components
- Logical grouping according to type of function

Various other issues are considered after service candidates are subjected to the service-oriented design process. For now, this grouping establishes a preliminary utility service layer in which utility service candidate capabilities are further associated with resources and methods. A primary input will be any utility-centric resources previously defined in Step 5.

Note

Modeling utility service candidates is notoriously more difficult than entity service candidates. Unlike entity services where we base functional contexts and boundaries upon already-documented enterprise business models and specifications (such as taxonomies, ontologies, entity relationships, etc.), there are usually no such models for application logic. Therefore, it is common for the functional scope and context of utility service candidates to be continually revised during iterations of the service inventory analysis cycle.

Case Study Example

The MUA team proceeds by digging up notes from prior process steps regarding utility-centric actions that have been documented so far. Combined with the research they collected from the Analyze Processing Requirements step, they proceed to define the following two utility services.

Notification Service Candidate

The Send Rejection Notice and Send Acceptance Notice actions are combined into one generic Send service capability candidate as part of a utility service called Notification (Figure 7.12). The Send capability will accept a range of input values, enabling it to issue approval and rejection notifications, among others.

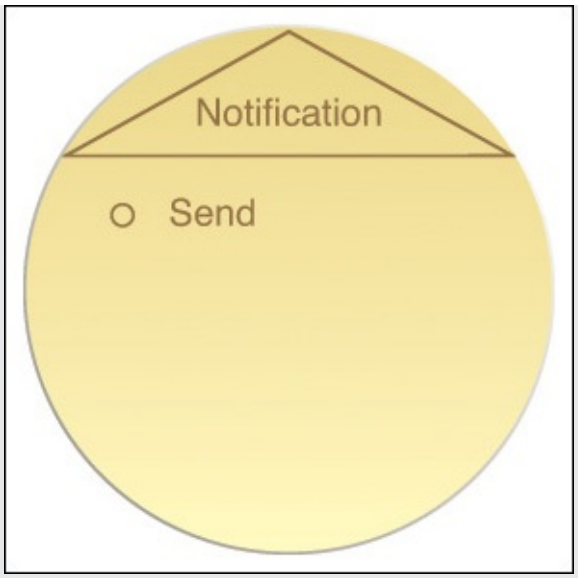


Figure 7.12 The Notification service candidate, with a sole service capability candidate that processes two of the actions identified for the parent business process.

Document Service Candidate

The MUA service modeling team originally created a Document Printing utility service, but then realized its functional scope was too limiting. Instead, it broadened its scope to encompass generic document processing functions. For the time being, this service candidate will only include a Print service capability candidate to accommodate the Print Hard Copy of Award Conferral Record action (Figure 7.13). In the future, this utility service will include other service capabilities that perform generic document processing tasks, such as faxing, routing, and parsing.

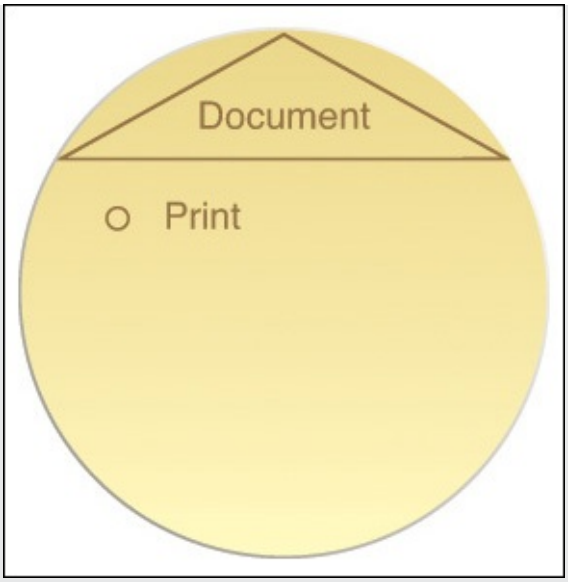


Figure 7.13 The Document service candidate with a generic Print service capability candidate.

Next, the *Notice Sender* and *Printer* resources identified earlier in Step 5 are revisited so that they, together with the appropriate methods, can be allocated to the newly defined utility service candidate capabilities.

Notification Service Candidate

The Send service capability candidate is expanded with the POST method and the *Notice Sender* resource (Figure 7.14).

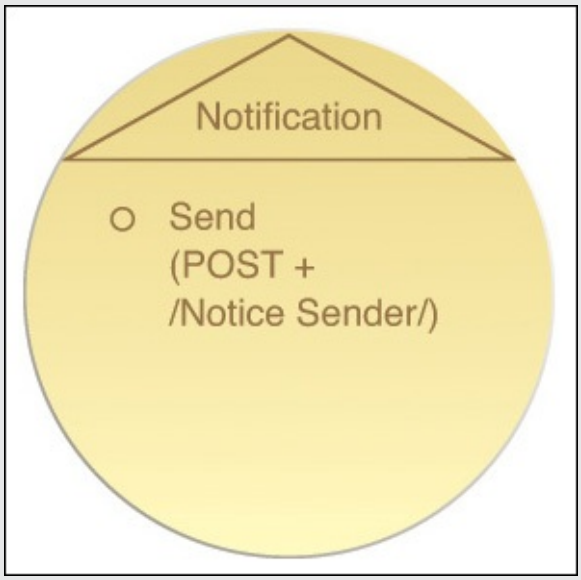


Figure 7.14 The Notification service candidate with method and resource association.

Document Service Candidate

The highly generic Print service capability candidate is expanded with a POST method and the *Printer* resource (Figure 7.15). Any document sent to the Print capability will be posted to the *Printer* resource and then printed.

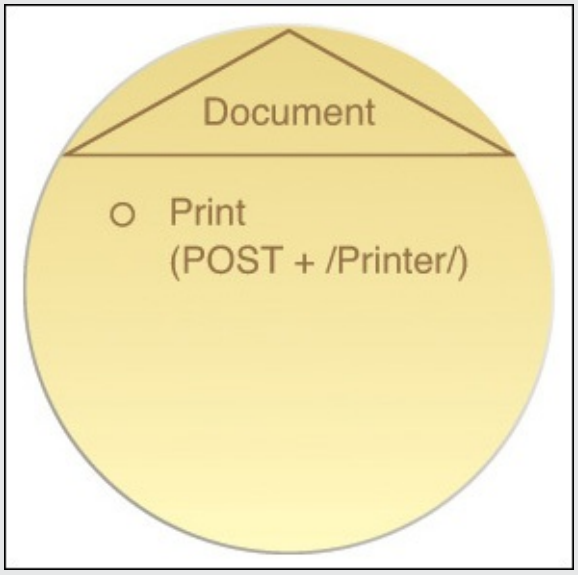


Figure 7.15 The Document service candidate with method and resource association.

Step 11: Define Microservice Candidates (and Associate Resources and Methods)

We now turn our attention to the previously identified non-agnostic processing logic to determine whether any unit of this logic may qualify for encapsulation by a separate microservice. As discussed in Chapter 5, the microservice model can introduce a highly independent and autonomous service implementation architecture that can be suitable for units of logic with particular processing demands.

Typical considerations can include:

- Increased autonomy requirements
- Specific runtime performance requirements
- Specific runtime reliability or failover requirements
- Specific service versioning and deployment requirements

Case Study Example

In support of isolating the processing for the Verify Student Transcript Qualifies for Award Based on Award Conferral Rules action, the MUA team establishes a microservice candidate called Verify Application, with a single Verify service capability candidate (Figure 7.16).

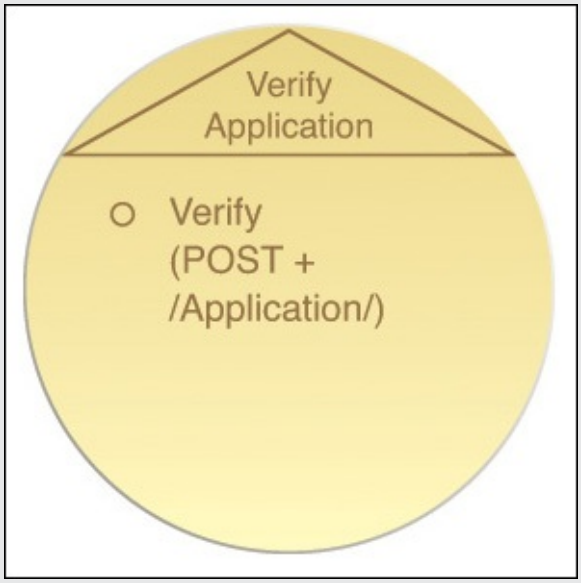


Figure 7.16 The Verify Application service candidate with method and resource association.

Verify Application Service

It is presumed that the eventual implementation environment for this service will be highly autonomous and may include a redundant implementation of the Rules service to guarantee the previously identified reliability requirements.

Step 12: Apply Service-Orientation

This step is a repeat of Step 7 provided here specifically for any new utility service candidates that may have emerged from the completion of Steps 9 and 10.

Step 13: Revise Candidate Service Compositions

Now we revisit the original service composition candidate scenarios we identified in Step 8 to incorporate new or revised utility service candidates. The result is typically an expansion of the service composition scope where more utility service capabilities find themselves participating in the business process

automation.

Case Study Example

The Confer Student Award service composition expands with the introduction of the Notification and Document utility services and the Verify Application microservice (Figure 7.17).

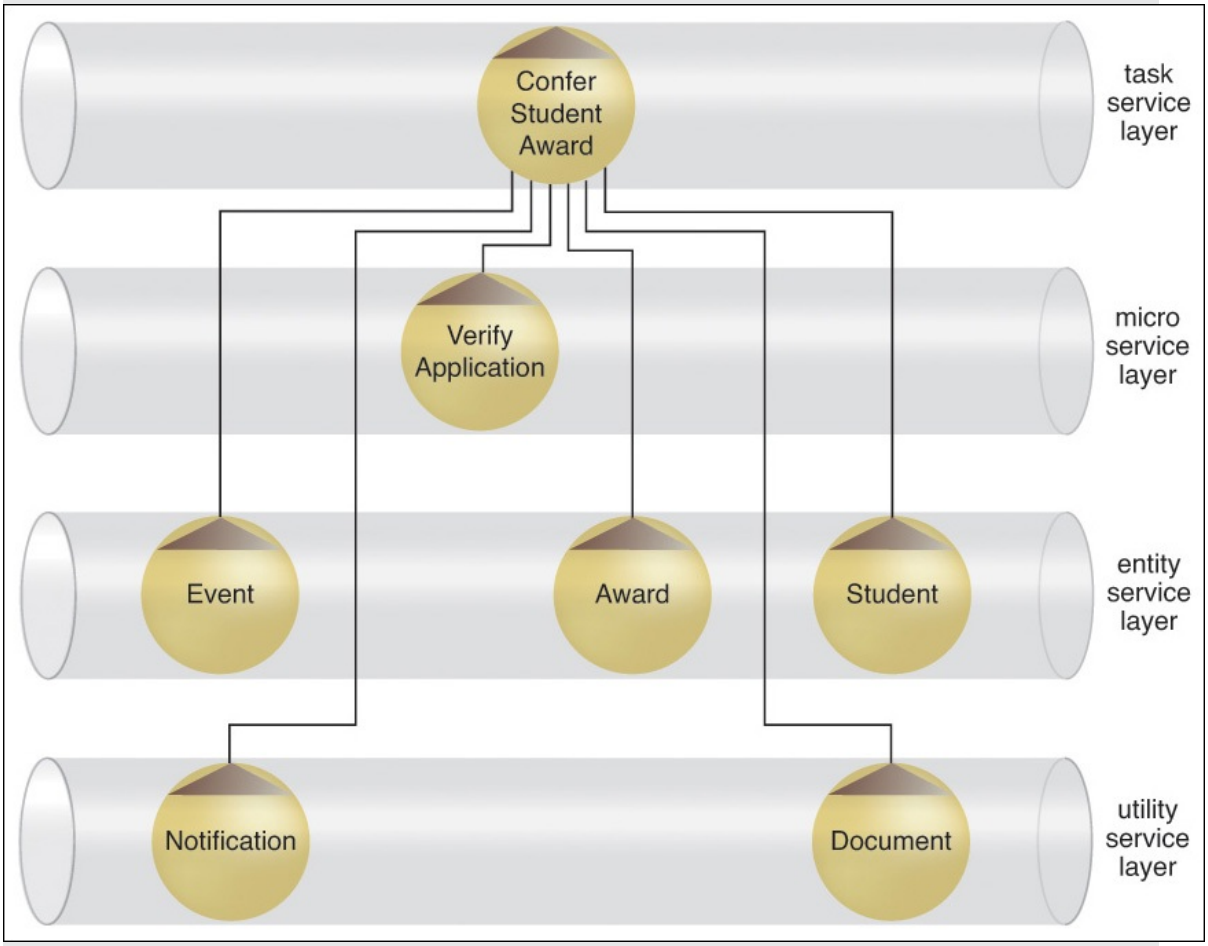


Figure 7.17 The revised service composition candidate incorporating new utility services and a microservice.

Step 14: Revise Resource Definitions and Capability Candidate Grouping

Both business-centric and utility-centric resources can be accessed or processed by utility services and microservices. Therefore, any new processing logic identified in the preceding steps can result in opportunities to further add to and/or revise the set of resources modeled so far.

Furthermore, with the introduction of new utility services and/or microservices, we need to check the grouping of all modeled service capability candidates because:

- Utility service capability candidates defined in Steps 9 and 10 may remove some of the required actions that comprised entity service capability candidates defined earlier, in Step 3.
- The introduction of new utility service candidates may affect (or assimilate) the functional scopes of already-defined utility service candidates.
- The modeling of larger and potentially more complex service composition candidates in Step 13 may lead to the need to reduce or increase the granularity of some service capability candidates.

Note

As a result, subsequent execution of several of the modeling steps will require an extra discovery task during which we determine what relevant service candidates, resources, and uniform contract properties exist, prior to defining or proposing new ones.

7.2 Additional Considerations

Uniform Contract Modeling and REST Service Inventory Modeling

A service inventory is a collection of services that are independently owned, governed, and standardized. When we apply the Uniform Contract {311} constraint during an SOA project, we typically do so for a specific service inventory. This is because a uniform contract will end up standardizing a number of aspects pertaining to service capability representation, data representation, message exchange, and message processing. The definition of a uniform contract is ideally performed prior to individual REST service contract design, because each REST service contract will be required to form dependencies on and operate within the scope of the features offered by its associated uniform contract.

Organizations that aim to build a single inventory of REST services will typically rely on a single overarching uniform contract to establish baseline communication standards. Those that proceed with a domain-based service

inventory approach instead will most likely need to define a separate uniform contract for each domain service inventory. Because domain service inventories tend to vary in terms of standardization and governance, separate uniform contracts can be created to accommodate these individual requirements. This is why uniform contract modeling can be part of the service inventory analysis project stage.

The purpose of the service inventory analysis stage is to enable a project team to first define the scope of a service inventory via the authoring of a service inventory blueprint. This specification is populated by the repeated execution of the service inventory analysis cycle. Once all iterations (or as many as are allowed) are completed, we have a set of service candidates that have been (hopefully) well-defined, both individually and in relation to each other. The subsequent step is to proceed with the design of the respective service contracts.

When we know in advance that we will be delivering these services using REST, it is beneficial to incorporate the modeling of the inventory's uniform contract into the modeling of the service inventory itself. This is because as we perform each service-oriented analysis process and model and refine each service candidate and service capability candidate, we gather more and more intelligence about the business automation requirements that are distinct to that service inventory. Some of this information will be relevant to how we define the methods and media types of the uniform contract.

Examples of useful areas of intelligence include:

- Understanding the types of information and documents that will need to be exchanged and processed can help define necessary media types.
- Understanding the service models (entity, utility, task, etc.) in use by service candidates can help determine which available methods should be supported.
- Understanding policies and rules that are required to regulate certain types of interaction can help determine when certain methods should not be used, or help define special features that may be required by some methods.
- Understanding how service capability candidates may need to be composed can help determine suitable methods.
- Understanding certain quality-of-service requirements (especially in relation to reliability, security, transactions, etc.) can help determine the need to support special features of methods, and may further help identify the need to issue a set of pre-defined messages that can be standardized as

complex methods.

A practical means of incorporating the task of uniform contract modeling as part of the service inventory analysis is to group it with the Define Technology Architecture step (Figure 7.18). During this step general service inventory architecture characteristics and requirements are identified from the same types of intelligence we collect for the definition of uniform contract features. In this context, the uniform contract is essentially being defined as an extension to the standardized technology architecture for the service inventory.

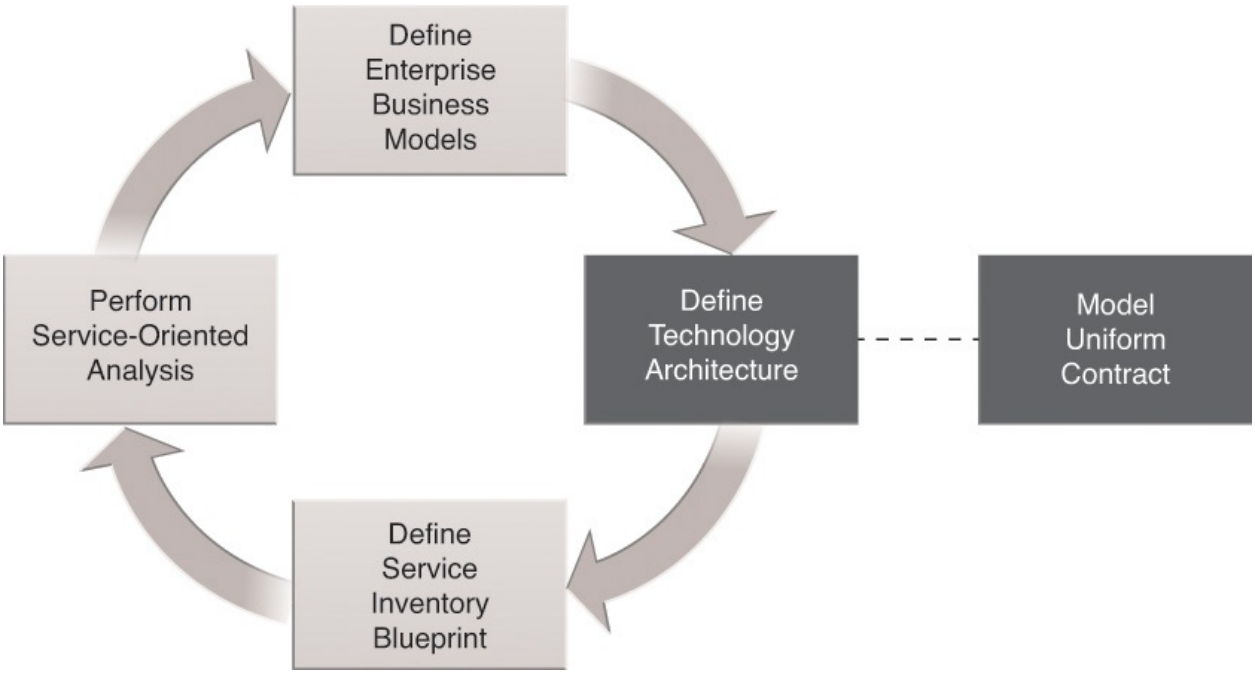


Figure 7.18 In the service inventory analysis cycle, uniform contract modeling can be included as an iterative task.

If combining the Model Uniform Contract task with the Define Technology Architecture step turns out to be an unsuitable grouping, then the Model Uniform Contract task can be positioned as its own step within the cycle.

When we begin working on the uniform contract definition, one of the key decisions will be to determine the sources to be used to populate its methods and media types. As a general starting point, we can look to the HTTP specification for an initial set of methods and the IANA Media Type Registry for the initial media types. Further media types and possibly further methods may come from a variety of internal and external sources.

Note

It is also worth noting that methods and media types can be

standardized independently of a service inventory. For example, HTTP methods are defined by the IETF. A service inventory that uses these methods will include a reference to the IETF specification as part of the service inventory uniform contract definition. Media types may be specified on an ongoing basis by external bodies, such as the W3C, the IETF, industry bodies across various supply chains, or even within an IT enterprise.

Note that the asterisk symbol can be used in the top-right corner to indicate that a REST service candidate is being modeled during this step that either:

- Incorporates methods and/or media types already modeled for the uniform contract, or
- Introduces the need to add or augment methods and/or media types for the uniform contract

This type of two-way relationship between the Perform Service-Oriented Analysis step (which encompasses the REST service modeling process) and the Model Uniform Contract task is a natural dynamic of the service inventory analysis cycle.

Note

It is usually during the Model Uniform Contract task that a uniform contract profile is first populated with preliminary characteristics and properties. This profile document is then further refined as the uniform contract and is physically designed and maintained over time.

REST Constraints and Uniform Contract Modeling

Although REST constraints are primarily applied during the physical design of service architectures, taking them into consideration as the uniform contract takes shape during the service-oriented analysis stage can be helpful. For example:

- *Stateless {308}* – From the data exchange requirements we are able to model between service candidates, can we determine whether services will be able to remain stateless between requests?
- *Cache {310}* – Are we able to identify any request messages with responses that can be cached and returned for subsequent requests instead

of needing to be processed redundantly?

- *Uniform Contract {311}* – Can all methods and media types we are associating with the uniform contract during this stage be genuinely reused by service candidates?
- *Layered System {313}* – Do we know enough about the underlying technology architecture to determine whether services and their consumers can tell the difference between communicating directly or communicating via intermediary middleware?

The extent to which concrete aspects of REST constraint application can be factored into how we model the uniform contract will depend directly on:

- The extent to which the service inventory technology architecture is defined during iterations of the service inventory analysis cycle, and
- The extent to which we learn about a given business process's underlying automation requirements during Step 2 of the service-oriented analysis process

Much of this will be dependent on the amount of information we have and are able to gather about the underlying infrastructure and overall ecosystem in which the inventory of services will reside. For example, if we know in advance that we are delivering a set of services within an environment riddled with existing legacy systems and middleware, we will be able to gain access to many information sources that will help determine boundaries, limitations, and options when it comes to service and uniform contract definition. On the other hand, if we are planning to build a brand-new environment for our service inventory, there will usually be many more options for creating and tuning the technology architecture in support of how the services (and the uniform contract) can best fulfill business automation requirements.

SOA Patterns

When determining the scope of a service inventory and whether multiple service inventories are allowed within an enterprise environment, the decision usually comes down to whether the Enterprise Inventory [340] or the Domain Inventory [338] pattern is applied.

REST Service Capability Granularity

When actions are defined at this stage, they are considered fine-grained in that

each action is clearly distinguished with a specific purpose. However, within the scope of that purpose they can often still be somewhat vague and can easily encompass a range of possible variations.

Defining conceptual service candidates using this level of action granularity is common with mainstream service modeling approaches. It has proven sufficient for SOAP-based Web services because service capabilities that need to support variations of functionality can still be effectively mapped to WSDL-based operations capable of handling a range of input and output parameters.

With REST service contracts, service capabilities are required to incorporate methods (and media types) defined by an overarching uniform contract. As already discussed in the preceding section, the uniform contract for a given service inventory can be modeled alongside and in collaboration with the modeling of service candidates, as long as we know in advance that REST will act as the primary service implementation medium.

Whereas a WSDL-based service contract can incorporate custom parameter lists and other service-specific features, REST puts an upper bound on the granularity of message exchanges at the level of the most complex or most general purpose method and media type. This may, in some cases, lead to the need to define finer-grained service capabilities.

Figure 7.19 highlights the difference between a service candidate modeled in an implementation-neutral manner versus one modeled specifically for the REST service implementation medium.

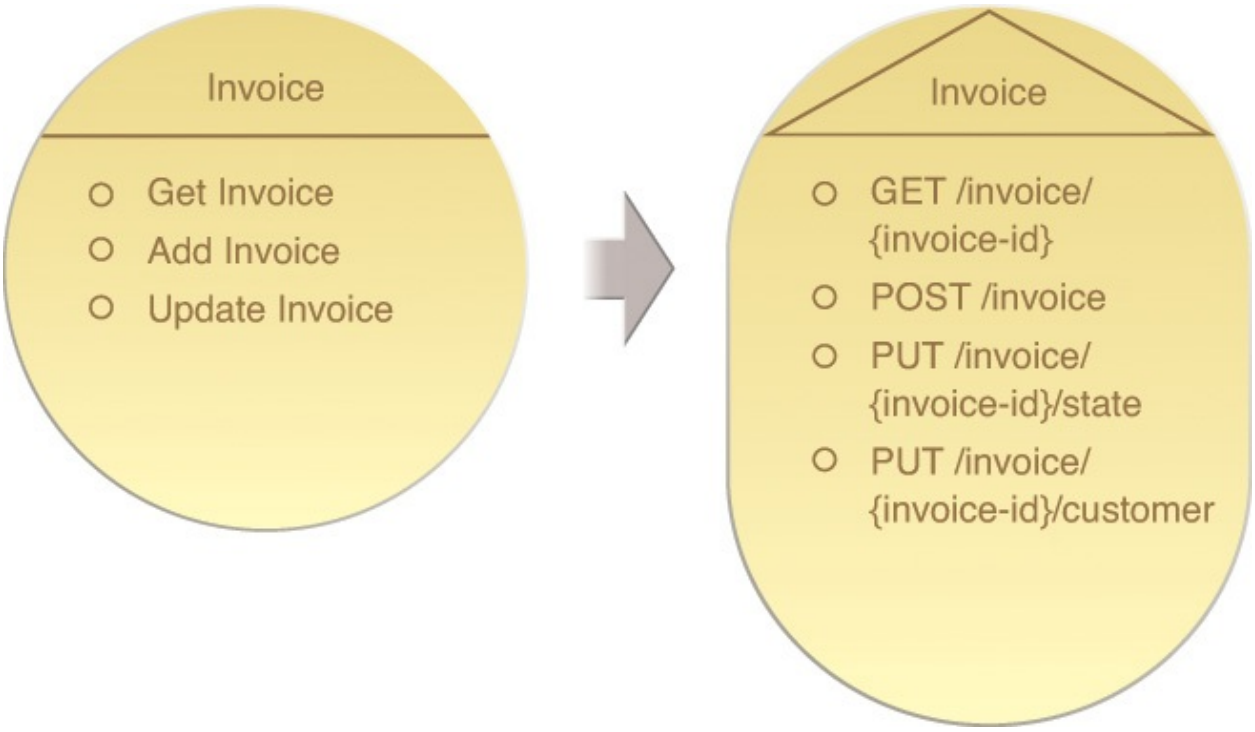


Figure 7.19 A REST service candidate can be modeled specifically to incorporate uniform contract characteristics. The Update Invoice service capability candidate is split into two variations of the PUT *invoice* service capability: one that updates the invoice state value, and another that updates the invoice customer value.

Resources vs. Entities

Part of the REST service modeling process explores the identification of resource candidates. It is through the definition of these resource candidates that we begin to introduce a Web-centric view of a service inventory. Resources represent the “things” that need to be accessed and processed by service consumers.

What we are also interested in establishing during the service-oriented analysis stage is the encapsulation of entity logic. As with resources, entities also often represent “things” that need to be accessed and processed by service consumers.

What then is the difference between a resource and an entity? To understand REST service modeling, we need to clearly understand this distinction:

- Entities are business-centric and are derived from enterprise business models, such as entity relationship diagrams, logical data models, and ontologies.
- Resources can be business-centric or non-business-centric. A resource is

any given “thing” associated with the business automation logic enabled by the service inventory.

- Entities are commonly limited to business artifacts and documents, such as invoices, claims, customers, *etc.*
- Some entities are more coarse-grained than others. Some entities can encapsulate others. For example, an invoice entity may encapsulate an invoice detail entity.
- Resources can also vary in granularity, but are often fine-grained. It is less common to have formally defined coarse-grained resources that encapsulate fine-grained resources.
- All entities can relate to or be based on resources. Not all resources can be associated with entities because some resources are non-business-centric.

The extent to which we need to formalize the mapping between business-centric resources and entities is up to us. The REST service modeling process provides steps that encourage us to define and standardize resources as part of the service inventory blueprint so that we gain a better understanding of how and where resources need to be consumed.

From a pure modeling perspective we are further encouraged to relate business-centric resources to business entities so that we maintain a constant alignment with how business-centric artifacts and documents exist within our business. This perspective is especially valuable as the business and its automation requirements continue to evolve over time.

Chapter 8. Service API and Contract Design with Web Services

Note

Parts of this chapter refer to the WSDL, SOAP, and XML Schema

markup languages and provide code examples. To learn about these and other Web services markup languages, see the *Web Service Contract Design and Versioning for SOA* series book.

After conceptual service candidates have been modeled and sufficiently refined, we reach the service-oriented design stage where we can begin designing physical service contracts based on the results of the preceding service-oriented analysis process.

When building SOAP-based Web services, this stage requires us to apply several contract-related service-orientation principles that help shape the design of the API as part of each service contract in a consistent and standardized manner prior to the design of the corresponding service logic.

Specifically, the following benefits can be attained via a contract-first approach with Web services:

- Web service contracts can be designed to accurately represent the context and function of their corresponding service candidates.
- Conventions can be applied to Web service operation names to produce standardized endpoint definitions.
- The granularity of operations can be modeled in abstract to provide consistent and predictable API designs that also establish a message size and volume ratio suitable for the target communications infrastructure.
- Service consumers are required to conform to the expression of the service contract, not vice versa.
- The design of business-centric Web service contracts can be assisted by business analysts who may be able to help establish an accurate expression of business logic.

We generally begin a Web service contract design with a formal definition of the messages the service is required to process. To accomplish this we need to formalize the message structures that are defined within the WSDL `types` area. SOAP messages carry payload data within the `Body` section of the SOAP envelope and this data needs to be organized and typed. For this we normally rely on XML schemas.

Note that during the service-oriented analysis process it may have been determined that one or more service candidates are more suitable for implementation via REST instead of the SOAP-based Web services technology set. This may be the case if microservices were identified, or other services that

have processing requirements better fulfilled via REST. For those service candidates, the service-oriented design guidelines covered in Chapter 9 are applied.

SOA Patterns

Service-oriented architectures can allow services within a single service inventory to be implemented via different communication protocols, as per the Dual Protocols [339] pattern. Additionally, as per the Concurrent Contracts [332] pattern, a single body of service logic can expose two alternative service contracts that allow it to be invoked via two different communication protocols. In support of this functionality, the Service Façade [360] pattern is often also applied together with Decoupled Contract [337].

8.1 Service Model Design Considerations

The choice of service model for a given service can affect our approach to Web service contract design. The following sections briefly raise some key considerations for each service model.

Entity Service Design

Entity services represent the one service layer that is the least influenced by others. Its purpose is to accurately represent corresponding data entities defined within an organization's business models. These services are business process–agnostic, built for reuse by any services within the same service inventory that may need to access or manage information associated with a particular entity. Because they exist rather independently in relation to other service layers, it is beneficial to design entity services prior to others. This establishes an abstract service layer around which process and underlying application logic can be positioned.

The Service Reusability (295) and Service Autonomy (297) principles are somewhat naturally part of the entity design model in that the operations exposed by entity services are intended to be inherently generic and reusable (and because the use of the `import` statement is encouraged to reuse schemas and create modular WSDL definitions).

Discoverability is also an important part of both the design of entity services and their post-deployment utilization, as we need to ensure that a service design does not implement logic already existing. A discovery mechanism would make this

determination easier. One measure we can take to make a service more discoverable to others is to supplement it with metadata details using the `documentation` element.

Figure 8.1 shows a sample entity Web service contract.

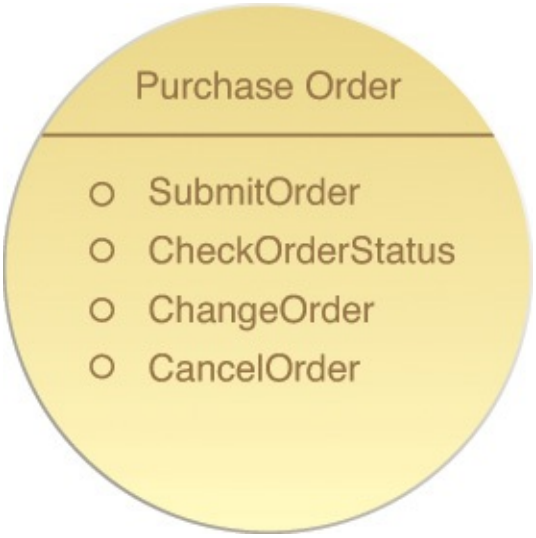


Figure 8.1 A sample entity service with four operations dedicated to functions pertaining to purchase order processing.

SOA Patterns

Due to the fact that entity services naturally process key business documents, the use of standardized XML schemas becomes a paramount design concern. This greatly emphasizes the need to enforce the application of the Canonical Schema [326] and Schema Centralization [356] patterns to all entity services within a service inventory.

Utility Service Design

Utility services are responsible for carrying out a variety of low-level processing functions. The SOAP-based Web services implementation option is suitable for utility services that need to expose a rich, well-defined API.

Unlike services in entity layers, the design of utility services does not require business analysis expertise. Utility Web services are generally an abstraction of portions of an organization's legacy environment, best defined by those who understand these environments the most.

Because of the real-world and technology-specific considerations that need to be

taken into account, utility services can be the hardest type of service to design. In addition, the context established by these services can be constantly challenged whenever technology is upgraded or replaced and related application logic built or altered.

The type of processing logic that resides in utility services can be similar to the type of logic placed in microservices. Both of these services commonly perform utility-centric processing. However, because utility services are agnostic, the Service Reusability (295) principle is a constant influence in how the service capabilities are designed, requiring the API to be as generic and flexible as possible. This consideration further carries over to determining the appropriate granularity of a given operation.

Furthermore, it is important to ensure that any newly defined agnostic utility functionality does not, in some way, shape, or form, already exist. It is therefore necessary to review the existing service inventory for services that may already resemble what is planned for a new utility service. Additionally, because these services provide such generic functionality, it is worth, at this stage, investigating whether the features you require can be purchased or leased from third-party vendors, as long as required quality of service levels can be met.

Figure 8.2 displays a simple utility Web service contract.

Figure 8.2 A sample utility service with a functional context dedicated to data transformation. The initial two operations are labeled specifically in relation to accounting data transformation to allow future transformation-style operations that may not be related to accounting data to be added.

SOA Patterns

Utility services are more likely to warrant support for alternative communication protocols, which makes the application of the Dual Protocols [339], Concurrent Contracts [332], and Service Façade [360] patterns more likely than with entity services. Another pattern commonly applied during the utility service contract design stage is Legacy Wrapper [347] for utility services dedicated to encapsulating legacy APIs.

In IT enterprises that have applied Domain Inventory [338], there is also the application of the Cross-Domain Utility Layer [336] pattern that can be considered, in order to leverage reuse opportunities.

Microservice Design

Although building a microservice as a SOAP-based Web service is possible, it is not a common approach. The processing overhead associated with SOAP messaging and the multilayered technology stack of Web service and WS-* environments can impose latency and other performance-related challenges that oppose the typical high-performance design goals of microservices.

This book therefore primarily covers the service contract design of REST-based microservices, as explained further in Chapter 9. If you are considering building microservices using Web service technologies, many of the guidelines raised in Chapter 9 will still apply.

SOA Patterns

Visit the *Microservice Design* section in Chapter 9 for a list of patterns that may be applicable to microservice contracts and implementations.

Task Service Design

Task services typically contain embedded workflow logic used to coordinate an underlying service composition. Therefore, the process for designing task services usually requires less effort than for any of the preceding service models, simply because they often only require an operation used as a trigger for initiating the workflow logic.

Additional operations can be added to support asynchronous interactions. For example, tasks that involve human interaction or batch processing will retain the state of the ongoing business process between requests and can allow access to

this state by exposing service operations for this purpose.

Different modeling approaches can be used to accomplish this step, such as the use of sequence diagrams (Figures 8.3 and 8.4). The purpose of this exercise is to document each possible execution path, including all exception conditions. The resulting diagrams also will be useful input for subsequent test cases.

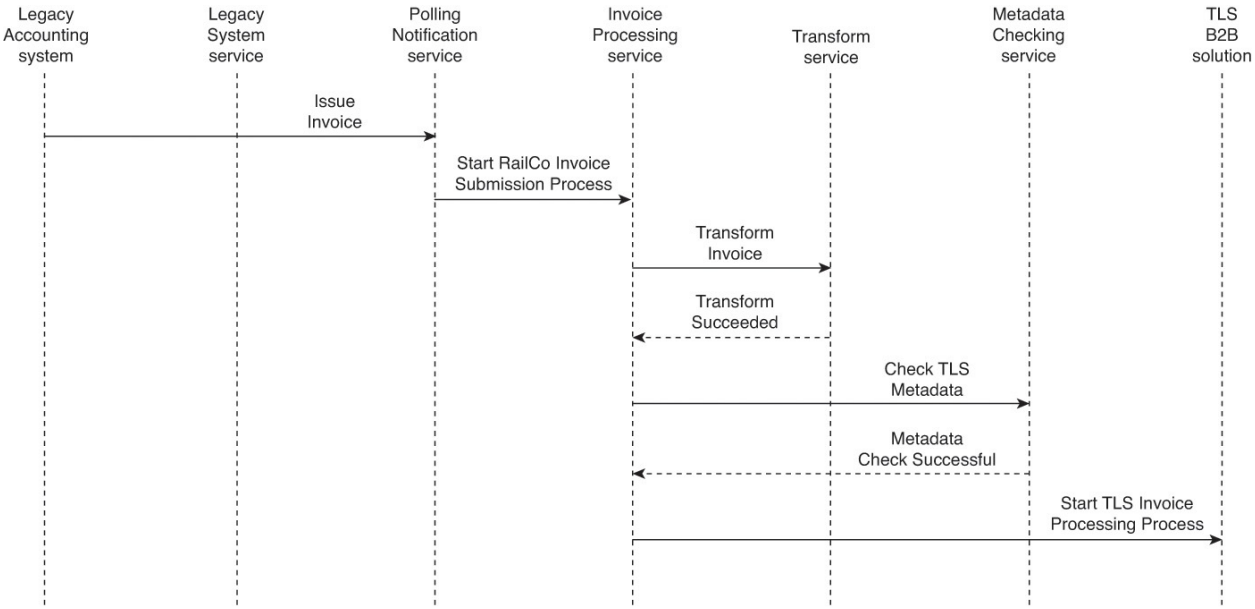


Figure 8.3 A successful completion of sample workflow logic carried out by a task service.

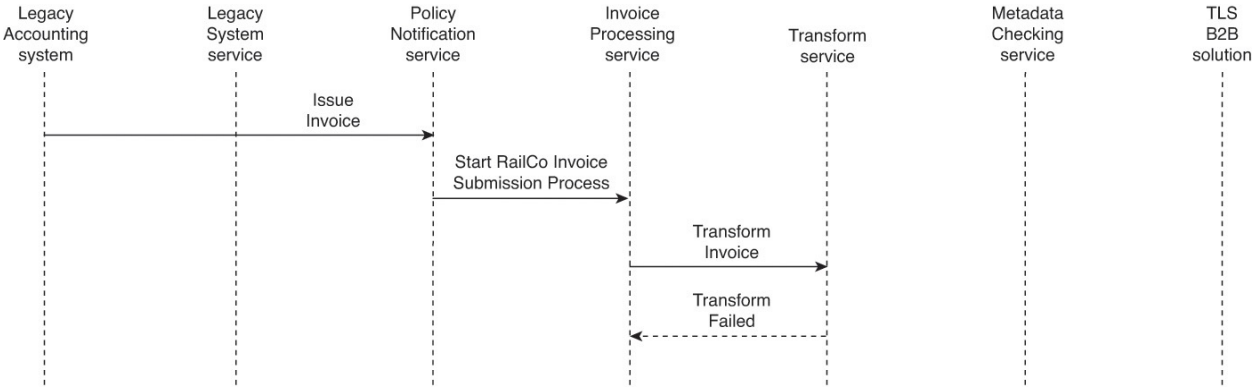


Figure 8.4 A failure condition caused by an error during the processing of sample workflow logic by a task service. In this case, one of its composed services returns an error that terminates the execution of the business process.

The workflow logic that task services can contain will frequently impose processing dependencies in service compositions. This can lead to the need for state management. However, the use of document-style SOAP messages may allow the task service to delegate the persistence of some or all of this state information to the message itself.

A task service with a single operation is shown in Figure 8.5.

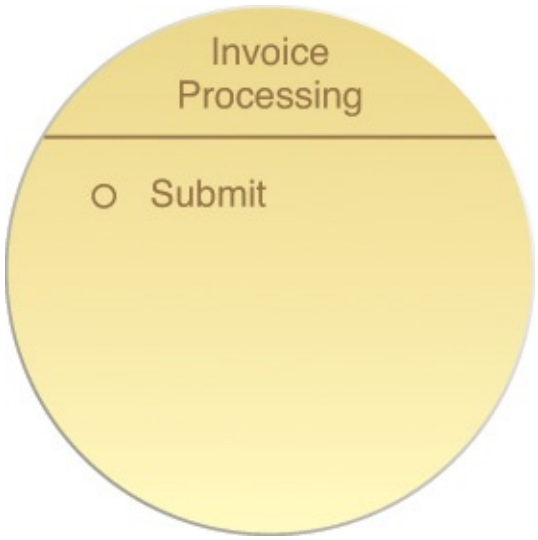


Figure 8.5 A sample task service that kicks off invoice processing workflow logic via a single Submit operation that receives an invoice document as input.

SOA Patterns

The workflow logic encapsulated by orchestrated task services may require the need to incorporate atomic transactions or orchestration and compensation type functionality, which corresponds to the use of the Atomic Service Transaction [324] and Compensating Service Transaction [330] patterns, respectively.

Several patterns exist to enable state management and support the application of the Service Statelessness (298) principle, including State Repository [363] and Partial State Deferral [352]. Furthermore, the State Messaging [362] pattern formalizes the aforementioned deferral of state information to the messaging layer, as enabled by SOAP messages.

Case Study Example

The service modeling exercise performed by TLS produced a number of Web service candidates in support of its new Timesheet Submission solution. The contract design of the Employee service is explored in this case study example. Figure 8.6 shows the original service candidate modeled in Chapter 6.

Figure 8.6 The Employee service candidate.

The Employee service was modeled in support of carrying out two specific functions:

- Executing a query against the employee record to retrieve the maximum number of hours the employee is authorized to work within a week.
- Post updates to the employee's history (required only when a timesheet is rejected).

TLS invested in creating a standardized XML Schema data representation architecture (for its accounting environment only) some time ago. As a result, a collection of entity XML schemas representing accounting-related information sets already exists.

At first, this appears to make this step rather simple. However, upon closer study, it is discovered that the existing XML schema is very large and complex. After some discussion, TLS architects decide that they will not use the existing schema with this service at this point. Instead, they opt to derive a lightweight (but still fully compliant) version of the schema to accommodate the simple processing requirements of the Employee service.

They begin by identifying the kinds of data that will need to be exchanged to fulfill the processing requirements of the Get Weekly Hours Limit capability candidate. They end up defining two complex types:

- One containing the search criteria required for the request message received by the Employee service
- One containing the query results returned by the service

The types are deliberately named so that they are associated with the respective messages. These two types then constitute the new Employee.xsd schema file, as shown in Example 8.1.

Example 8.1 The Employee schema providing `complexType` constructs used to establish the data representation anticipated for the Get Weekly Hours Limit capability candidate.

Click here to view code image

```
<xml:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace=
    "http://www.example.org/tls/employee/schema/accounting/">
  <xml:element name="EmployeeHoursRequestType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ID" type="xml:integer"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
  <xml:element name="EmployeeHoursResponseType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ID" type="xml:integer"/>
        <xml:element name="WeeklyHoursLimit"
          type="xml:short"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
</xml:schema>
```

However, just as the architects attempt to derive the types required for the Update Employee History capability candidate, another problem presents itself. They discover that the schema from which they derived the Employee.xsd file does not represent the EmployeeHistory entity, which this service candidate also encapsulates.

Another visit to the accounting schema archive reveals that employee history information is not governed by the accounting solution. It is, instead, part of the HR environment, for which no schemas have been created.

Not wanting to impose on the already-standardized design of the Employee schema, it is decided that a second schema definition be created, named EmployeeHistory.xsd (Example 8.2 and Figure 8.7).

Example 8.2 The EmployeeHistory schema, with a different `targetNamespace` to identify its distinct origin.

Click here to view code image

```
<xml:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace=
    "http://www.example.org/tls/employee/schema/hr/">
  <xml:element name="EmployeeUpdateHistoryRequestType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ID" type="xml:integer"/>
        <xml:element name="Comment" type="xml:string"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
  <xml:element name="EmployeeUpdateHistoryResponseType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ResponseCode"
          type="xml:byte"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
</xml:schema>
```

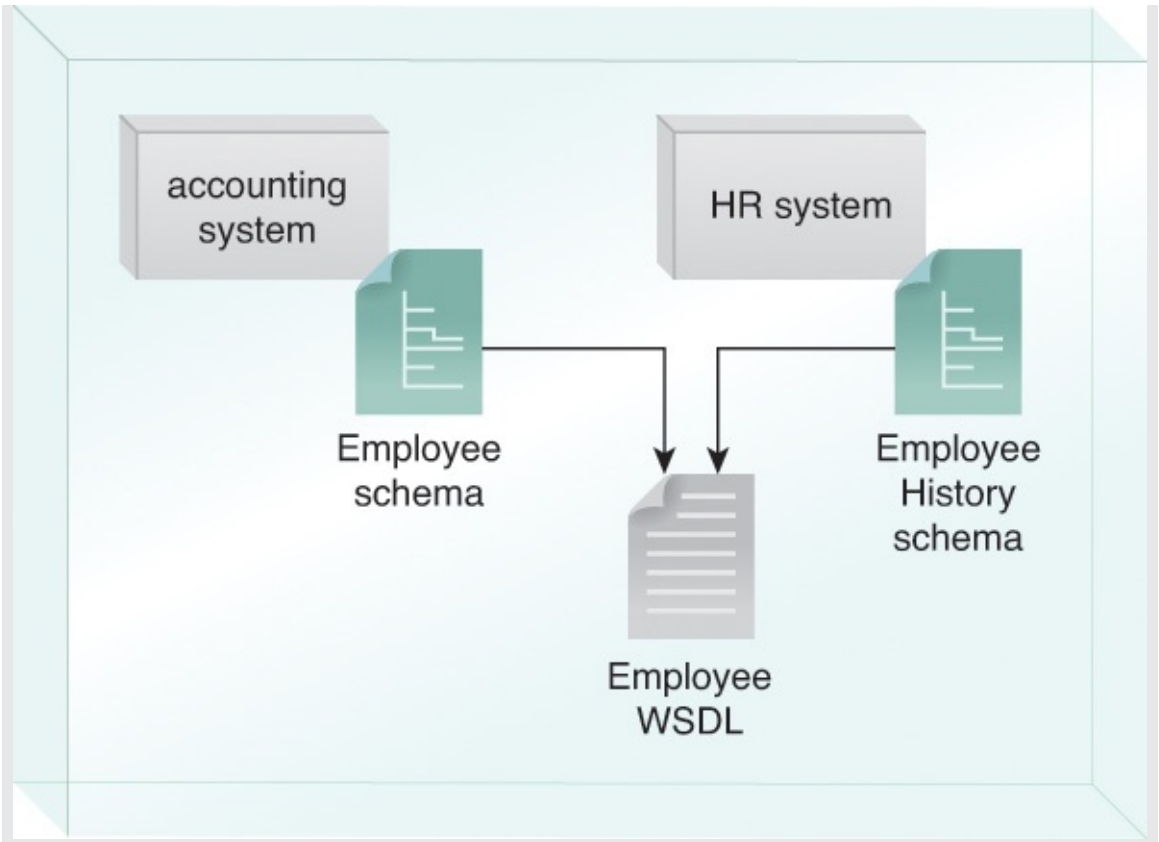


Figure 8.7 Two schemas originating from two different data sources.

To promote reusability and to allow for each schema file to be maintained separately from the WSDL definition, the XML Schema `import` statement is used to pull the contents of both schemas into the Employee service WSDL `types` construct (Example 8.3).

Example 8.3 The WSDL `types` construct being populated by imported schemas.

Click here to view code image

```
<types>
  <xml:schema targetNamespace=
    "http://www.example.org/tls/employee/schema/">
    <xml:import namespace=
      "http://www.example.org/tls/employee/schema/accounting/"
      schemaLocation="Employee.xsd"/>
    <xml:import namespace=
      "http://www.example.org/tls/employee/schema/hr/"
      schemaLocation="EmployeeHistory.xsd"/>
  </xml:schema>
</types>
```

Next, TLS architects follow these steps to define an initial service contract:

1. They confirm that each capability candidate is suitably generic and reusable by ensuring that the granularity of the logic encapsulated is appropriate. They then study the data structures defined earlier and establish a set of operation names.
2. They create the `portType` (or `interface`) area within the WSDL document and populate it with `operation` constructs that correspond to capability candidates.
3. They formalize the list of input and output values required to accommodate the processing of each operation's logic. This is accomplished by defining the appropriate `message` constructs that reference the XML Schema types within the child `part` elements.

The TLS architects decide on operation names *GetEmployeeWeeklyHoursLimit* and *UpdateEmployeeHistory* (Figure 8.8).

Figure 8.8 The Employee service operations.

They subsequently proceed to define the remaining parts of the abstract definition, namely the `message` and `portType` constructs, as shown in Example 8.4.

Example 8.4 The `message` and `portType` parts of the Employee service definition that implement the abstract definition details of

the two service operations.

Click here to view code image

```
<message name="getEmployeeWeeklyHoursRequestMessage">
  <part name="RequestParameter"
    element="act:EmployeeHoursRequestType"/>
</message>
<message name="getEmployeeWeeklyHoursResponseMessage">
  <part name="ResponseParameter"
    element="act:EmployeeHoursResponseType"/>
</message>
<message name="updateEmployeeHistoryRequestMessage">
  <part name="RequestParameter"
    element="hr:EmployeeUpdateHistoryRequestType"/>
</message>
<message name="updateEmployeeHistoryResponseMessage">
  <part name="ResponseParameter"
    element="hr:EmployeeUpdateHistoryResponseType"/>
</message>
<portType name="EmployeeInterface">
  <operation name="GetEmployeeWeeklyHoursLimit">
    <input message=
      "tns:getEmployeeWeeklyHoursRequestMessage"/>
    <output message=
      "tns:getEmployeeWeeklyHoursResponseMessage"/>
  </operation>
  <operation name="UpdateEmployeeHistory">
    <input message=
      "tns:updateEmployeeHistoryRequestMessage"/>
    <output message=
      "tns:updateEmployeeHistoryResponseMessage"/>
  </operation>
</portType>
```

Note

TLS has standardized on the WSDL 1.1 specification because it is conforming to the requirements dictated by version 1.1 of the WS-I Basic Profile and because none of its application platforms support a newer WSDL version. WSDL 1.1 uses the `portType` element instead of the `interface` element, which is provided by WSDL 2.0.

Upon a review of the initial abstract service interface, it is determined that a minor revision can be incorporated to better support fundamental service-orientation. Specifically, meta-information is added to the WSDL definition to better describe the purpose and function of each of the two

operations and their associated messages (Example 8.5).

Example 8.5 The service contract, supplemented with additional metadata documentation.

Click here to view code image

```
<portType name="EmployeeInterface">
  <documentation>
    GetEmployeeWeeklyHoursLimit uses the Employee
    ID value to retrieve the WeeklyHoursLimit value.
    UpdateEmployeeHistory uses the Employee ID value
    to update the Comment value of the EmployeeHistory.
  </documentation>
  <operation name="GetEmployeeWeeklyHoursLimit">
    <input message=
      "tns:getEmployeeWeeklyHoursRequestMessage"/>
    <output message=
      "tns:getEmployeeWeeklyHoursResponseMessage"/>
  </operation>
  <operation name="UpdateEmployeeHistory">
    <input message=
      "tns:updateEmployeeHistoryRequestMessage"/>
    <output message=
      "tns:updateEmployeeHistoryResponseMessage"/>
  </operation>
</portType>
```

The architect in charge of the Employee service design decides to make adjustments to the abstract service interface to apply current design standards. Specifically, naming conventions are incorporated to standardize operation names, as shown in Figure 8.9 and Example 8.6.

Figure 8.9 The revised Employee service operation names.

Example 8.6 The two `operation` constructs with new, standardized names.

Click here to view code image

```
<operation name="GetWeeklyHoursLimit">
  <input message="tns:getWeeklyHoursRequestMessage"/>
  <output message="tns:getWeeklyHoursResponseMessage"/>
</operation>
<operation name="UpdateHistory">
  <input message="tns:updateHistoryRequestMessage"/>
  <output message="tns:updateHistoryResponseMessage"/>
</operation>
```

Let’s take another look at the two operations that have been designed into the Employee service:

- GetWeeklyHoursLimit
- UpdateHistory

The first requires access to the employee profile. At TLS, employee information is stored in two locations:

- Payroll data is kept within the accounting system repository, along with additional employee contact information.
- Employee profile information, including employee history details, is stored in the HR repository.

When an XML Schema data representation architecture was first implemented at TLS, entity XML schemas were used to bridge

some of the existing disparity that existed among the many TLS data sources. Being aware of this, the architect investigates the origins of the Employee.xsd schema used as part of the Employee.wsdl definition to determine the processing requirements for the GetWeeklyHoursLimit operation.

It is discovered that although the schema accurately expresses a logical data entity, it represents a document structure derived from two different physical repositories. Subsequent analysis reveals that the weekly hours limit value is stored in the accounting database. The processing requirement for the GetWeeklyHoursLimit operation is then written up as follows:

Utility service-level function capable of issuing the following query against the accounting database: Return Employee's Weekly Hour Limit Using the Employee ID as the Only Search Criteria

Next, the details behind the UpdateHistory operation are studied. This time it's a bit easier, as the EmployeeHistory.xsd schema is associated with a single data source—the HR employee profile repository. Looking back at the original analysis documentation, the architect identifies the one piece of information that this particular solution will need to update within this repository. Therefore, the processing requirement definition goes beyond the immediate requirements of the solution, as follows:

Utility service-level function capable of issuing an update to the "comment" column of the employee history table in the HR employee profile database, using the employee ID value as the sole criteria.

At first glance, it looks like the Timesheet Submission solution may require new utility services to facilitate Employee service processing requirements, as illustrated in the expanded composition shown in Figure 8.10. These newly identified requirements will need to be subjected to the service modeling process described in Chapter 6.

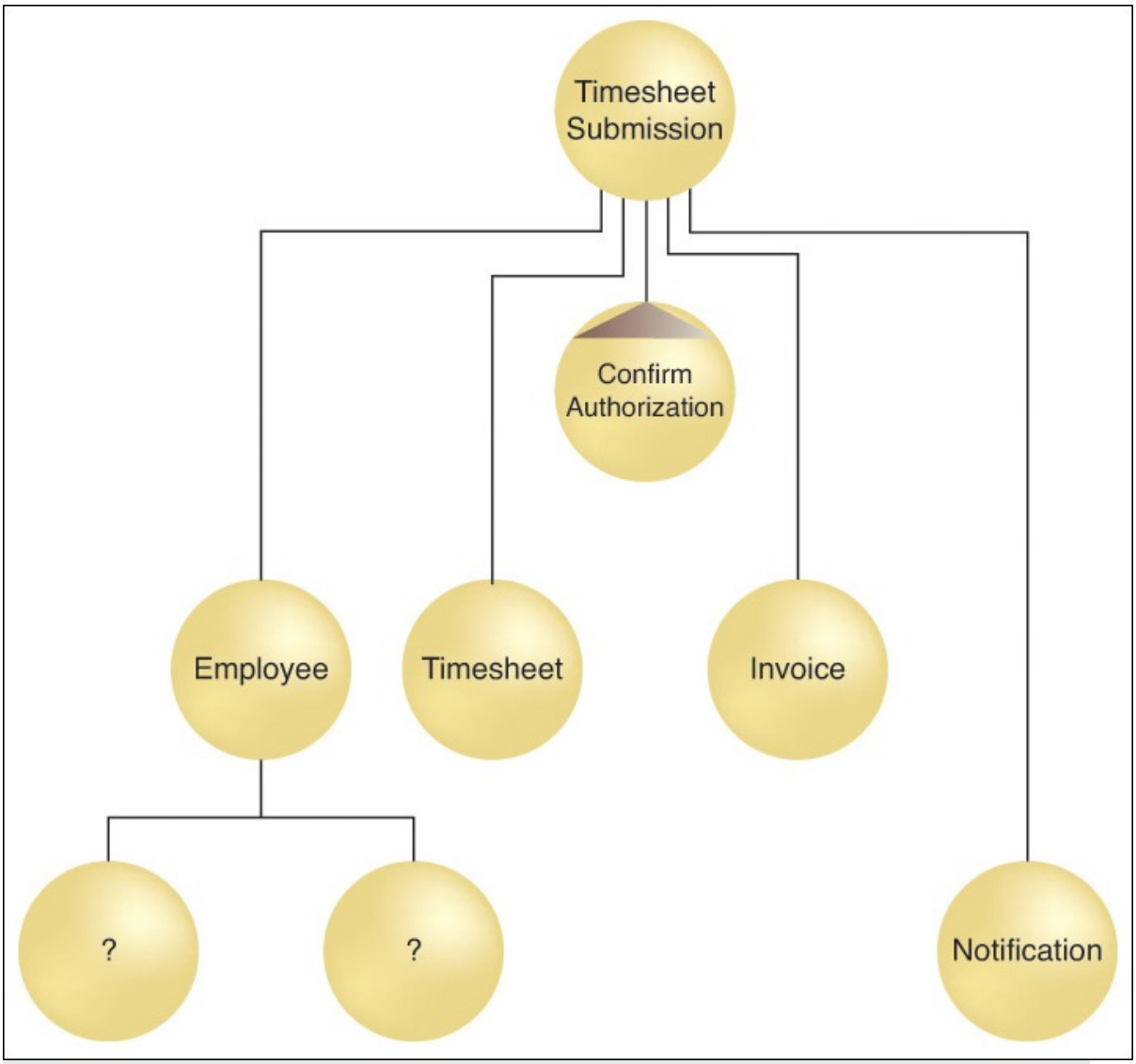


Figure 8.10 The revised composition hierarchy identifying new potential utility services.

It is eventually revealed that only one new utility service is required to accommodate the Employee service—a Human Resources wrapper service that also can facilitate the Timesheet service. Example 8.7 contains the final version of the Employee service definition, incorporating the changes to element names and the previous revisions.

Example 8.7 The final abstract service definition for the Employee service contract. The next step for this service will be to proceed with its concrete service definition and its service logic.

Click here to view code image

```
<definitions name="Employee"
  targetNamespace="http://www.example.org/tls/employee/wsdl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:act=
    "http://www.example.org/tls/employee/schema/accounting/"
  xmlns:hr="http://www.example.org/tls/employee/schema/hr/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.example.org/tls/employee/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <xml:schema targetNamespace=
      "http://www.example.org/tls/employee/schema/">
      <xml:import namespace=
        "http://www.example.org/tls/employee/schema/
          accounting/"
        schemaLocation="Employee.xsd"/>
      <xml:import namespace=
        "http://www.example.org/tls/employee/schema/hr/"
        schemaLocation="EmployeeHistory.xsd"/>
    </xml:schema>
  </types>
  <message name="getWeeklyHoursRequestMessage">
    <part name="RequestParameter"
      element="act:EmployeeHoursRequestType"/>
  </message>
  <message name="getWeeklyHoursResponseMessage">
    <part name="ResponseParameter"
      element="act:EmployeeHoursResponseType"/>
  </message>
  <message name="updateHistoryRequestMessage">
    <part name="RequestParameter"
      element="hr:EmployeeUpdateHistoryRequestType"/>
  </message>
  <message name="updateHistoryResponseMessage">
    <part name="ResponseParameter"
      element="hr:EmployeeUpdateHistoryResponseType"/>
  </message>
  <portType name="EmployeeInterface">
    <documentation>
      GetWeeklyHoursLimit uses the Employee ID value
      to retrieve the WeeklyHoursLimit value.
      UpdateHistory uses the Employee ID value to
      update the Comment value of the EmployeeHistory.
    </documentation>
    <operation name="GetWeeklyHoursLimit">
      <input message=
        "tns:getWeeklyHoursRequestMessage"/>
      <output message=
        "tns:getWeeklyHoursResponseMessage"/>
    </operation>
    <operation name="UpdateHistory">
```

```
        <input message=
          "tns:updateHistoryRequestMessage"/>
        <output message=
          "tns:updateHistoryResponseMessage"/>
      </operation>
    </portType>
    ...
  </definitions>
```

8.2 Web Service Design Guidelines

Provided in this section is a set of common guidelines for the design of Web service contracts. Several of these guidelines can become the basis of formal custom design standards.

Apply Naming Standards

Labeling Web services is the equivalent to labeling IT infrastructure. It is therefore essential that service APIs be as consistently self-descriptive as possible.

Naming standards therefore need to be defined and applied to:

- Service endpoint names
- Service operation names
- Message values

Here are some suggestions:

- Service candidates with high reuse potential should always be stripped of any naming characteristics that hint at the business processes for which they were originally built. For example, instead of naming an operation GetTimesheetSubmissionID, it can be simply reduced to GetTimesheetID or even just GetID.
- Entity services need to remain representative of the entity models from which their corresponding service candidates were derived. Therefore, the naming conventions used must reflect those established in the organization’s original entity models. Typically, this type of service uses the noun-only naming structure. Examples of suitable entity service names are Invoice, Customer, and Employee.
- Service operations for entity services should be verb-based and should not repeat the entity name. For example, an entity service called Invoice should not have an operation named AddInvoice.

- Utility services need to be named according to the processing context under which their operations are grouped. Both the verb+noun or noun only conventions can be used. Simplified examples of suitable utility service names are CustomerDataAccess, SalesReporting, and GetStatistics.
- Utility service operations need to clearly communicate the nature of their individual functionality. Examples of suitable utility service operation names are GetReport, ConvertCurrency, and VerifyData.
- While microservices are not always subjected to the same design standards as agnostic services, it is still recommended that the conventions for service and operation names be applied consistently to whatever extent possible.

Whatever naming standards are chosen, the key is that they must be consistently applied throughout all services within a given service inventory.

SOA Patterns

The Canonical Expression [325] pattern formalizes the use of naming conventions for standardization purposes.

Apply a Suitable Level of Contract API Granularity

When designing services, there are different granularity levels that need to be taken into consideration, as follows:

- *Service Granularity* – This represents the functional scope of a service. For example, fine-grained service granularity indicates that there is a small quantity of logic associated with the service's overall functional context.
- *Capability Granularity* – The functional scope of individual service capabilities is represented by this granularity level. For example, a GetDetail capability will tend to have a finer measure of granularity than a GetDocument capability.
- *Constraint Granularity* – The level of validation logic detail is measured by constraint granularity. For example, the coarser the constraint granularity, the less constraints (or smaller the amount of data validation logic) a given capability will have.
- *Data Granularity* – This granularity level represents the quantity of data processed. For example, a fine level of data granularity is equivalent to a small amount of data.

Because the level of service granularity determines the functional scope of a

service, it is usually determined during the analysis and modeling stages that precede service contract design. Once a service's functional scope has been established, the other granularity types come into play and affect both the modeling and physical design of the service contract (Figure 8.11).

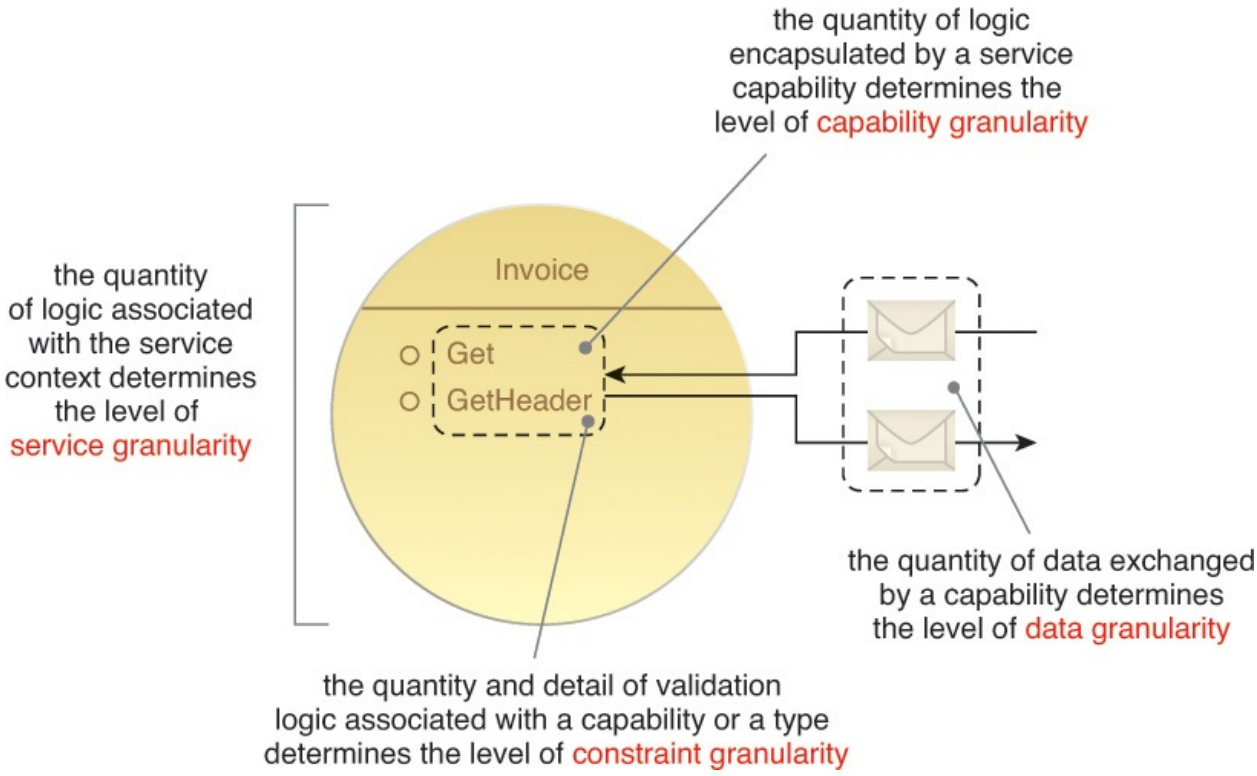


Figure 8.11 The four granularity levels that represent various characteristics of a service and its contract. Note that these granularity types are, for the most part, independent of each other.

Granularity is generally measured in terms of fine and coarse levels. It is worth acknowledging that the use of the terms *fine-grained* and *coarse-grained* is highly subjective. What may be fine-grained in one case may not be in another. The point is to understand how these terms can be applied when comparing parts of a service or when comparing services with each other.

Note

The term "constraint granularity" is not associated with the term *constraint* as it pertains to REST.

Although the granularity at which services can be designed can vary, there is a tendency to create APIs for Web services that are coarse-grained in order to get the most out of each message exchange. Performance, of course, is critical to the

success and ultimate evolution of service-oriented solutions. However, other considerations also need to be taken into account.

The coarser the granularity of a service contract, the less reuse it may be able to offer. If multiple functions are bundled into a single operation, it may be undesirable for consumers who only require the use of one of those functions. Additionally, some coarse-grained APIs may actually impose redundant processing or data exchange by forcing consumers to submit data not relevant to a particular activity.

Service contract granularity is a key strategic decision point that deserves a good deal of attention during the service-oriented design phase. Here are some guidelines for tackling this issue:

- Fully understand the performance limitations of the target deployment environment and explore alternative supporting technologies, if required.
- Explore the possibility of providing alternate (coarse and less coarse-grained) WSDL definitions for the same Web services. Or explore the option of supplying redundant coarse and less coarse-grained operations in the same WSDL definition. These approaches de-normalize service contracts but can address performance issues and accommodate a range of consumers.
- Assign coarse-grained APIs to services designated as solution endpoints and allow finer-grained APIs for services confined to pre-defined boundaries. This, of course, runs somewhat contrary to service-orientation principles and SOA characteristics that promote reuse and interoperability in services. Interoperability is promoted in coarse-grained services, and reusability is more fostered in finer-grained services.
- Consider the use of secondary service contracts that support alternative, more efficient communication protocols. Although it adds to its governance burden, it is possible to support a second communications medium within a service inventory. For example, it may be warranted to provide support for REST services alongside SOAP-based Web services.

Regardless of your approach, ensure that it is consistent and predictable so that an SOA can meet performance demands while remaining standardized.

Case Study Example

TLS chose an approach to contract API granularity where services positioned for use by consumers outside of TLS would provide consistently coarse-grained APIs. Operations on these services

would accept all the data required to process a particular activity. Further round-trips between external consumer and the service would only be required if absolutely necessary or if internal policies demanded it. Services used within TLS could provide less coarse-grained operations to facilitate reuse and a broader range of potential (internal) consumers, as long as the processing overhead imposed by less coarse-grained operations was acceptable.

SOA Patterns

Providing alternative contracts for the same service is addressed in the Concurrent Contracts [332] pattern. Adding redundant operations within the same Web service contract is formalized via the Contract Denormalization [335] pattern. Support for two communication protocols within the same service inventory is described in the Dual Protocols [339] pattern.

Design Web Service Operations to Be Inherently Extensible

Regardless of how well services are designed when first deployed, they can never be fully prepared for what the future holds. Some types of business process changes result in the need for the scope of entities to be broadened. As a result, corresponding business services may need to be extended. While the application of Service Reusability (295) and Service Composability (302) are thought through when partitioning logic as part of the service modeling process, extensibility is more of a physical design quality that needs to be considered during design.

Depending on the nature of the change, extensibility can sometimes be achieved without breaking the existing service contract. It is important to design Web service operations and messages to be as activity-agnostic as possible. This supports the processing of future non-specific values and functions that are still related to the operation's or message's overall purpose. Furthermore, it is a good habit to respond to new processing requirements by first investigating the possibility of composing other available services (including services that can be purchased or leased). This may succeed in fulfilling requirements without having to touch the service contract.

Note that extensions to an existing service contract will commonly impact its corresponding XML schema. These extensions can be facilitated by supplying

new schemas specifically for the extension. Before going down this road, though, ensure that established version control standards are firmly in place.

Case Study Example

Due to the size of TLS's organization, it is not uncommon for employees to be reallocated or to seek vertical or lateral position changes. The latter scenario is made further common by the "promote from within" motto encouraged by many divisional directors.

When an employee changes position or rank, the employee is expected to update his/her own profile using a form on the local intranet. Because this step is voluntary, it is often never performed. This, predictably, results in an increasingly out-of-date set of profiles. To counter this trend, the TLS Timesheet Submission process is altered to include an Employee Profile Verification step. When implemented, it will verify profile information prior to accepting a timesheet. Timesheets submitted by employees with invalid profiles will simply be rejected.

To implement this new requirement, the Timesheet service contract is not altered. Instead, the underlying service logic is extended to invoke a separate utility service that performs the profile verification.

SOA Patterns

An example of a pattern that can be applied to support future extensibility is Validation Abstraction [365], which decreases constraint granularity in order to support potential changes to validation logic.

Consider Using Modular WSDL Documents

WSDL service descriptions can be assembled dynamically at runtime through the use of import statements that link to separate files that contain parts of the service definition. This allows you to define modules for types, operations, and bindings that can be shared across WSDL documents.

It also allows you to leverage any existing XML Schema modules you may already have designed. You can separate schemas into granular modules that

represent individual complex types. This establishes a centralized repository of schemas that can be assembled into customized master schema definitions. By enabling you to import XML Schema modules into the `types` construct of a WSDL definition, you now can have your WSDL documents use those same schema modules.

Case Study Example

TLS considers importing the `bindings` construct so that it can be reused and perhaps even dynamically determined. However, it is later decided to leave the `bindings` construct as part of the WSDL document. Example 8.8 shows how the `import` statement is used to carry out this test.

Example 8.8 An `import` element used to pull in the `bindings` construct residing in a separate file.

Click here to view code image

```
<import namespace="http://.../common/wsdl/"
  location="http://.../common/wsdl/bindings.wsdl"/>
```

Use Namespaces Carefully

A WSDL definition consists of a collection of elements with different origins. Therefore, each definition often will involve a number of different namespaces. Following is a list of common namespaces used to represent specification-based elements:

Click here to view code image

```
http://schemas.xmlsoap.org/wsdl/
http://schemas.xmlsoap.org/wsdl/soap/
http://www.w3.org/2001/XMLSchema/
http://schemas.xmlsoap.org/wsdl/http/
http://schemas.xmlsoap.org/wsdl/mime/
http://schemas.xmlsoap.org/soap/envelope/
```

When assembling a WSDL from modules, additional namespaces come into play, especially when importing XML Schema definitions. Further, when defining your own elements, you can establish more namespaces to represent application-specific parts of the WSDL documents. It is not uncommon for larger WSDL documents to contain up to ten different namespaces and the qualifiers to go along with them. Therefore, it is highly recommended that you

organize the use of namespaces carefully within and across WSDL documents.

It is a common convention to require the use of the `targetNamespace` attribute to assign a namespace to the WSDL as a whole. If the XML schema is embedded within the WSDL definition, then it can also be assigned a `targetNamespace` value (which can be the same value used by the WSDL `targetNamespace`).

Case Study Example

Some of the common namespaces identified earlier are not required by the TLS Employee service and therefore are omitted from the list of `definitions` attributes. As shown in Example 8.9, the `targetNamespace` is added, along with two namespaces associated with the two imported schemas.

Example 8.9 The namespace declarations within the `definitions` element of the TLS Employee.wsdl file.

Click here to view code image

```
<definitions name="Employee"
  targetNamespace="http://www.xmltc.com/tls/employee/wsdl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:act=
    "http://www.xmltc.com/tls/employee/schema/accounting/"
  xmlns:hr="http://www.xmltc.com/tls/employee/schema/hr/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.xmltc.com/tls/employee/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ...
</definitions>
```

Use the SOAP Document and Literal Attribute Values

Two specific attributes establish the SOAP message payload format and the data type system used to represent payload data. These are the `style` attribute used by the `soap:binding` element and the `use` attribute assigned to the `soap:body` element. Both of these elements reside within the WSDL `binding` construct.

How these attributes are set is significant as it relates to the manner in which SOAP message content is structured and represented.

The `style` attribute can be assigned a value of “document” or “rpc.” The

former supports the embedding of entire XML documents within the SOAP body, whereas the latter is designed more to mirror traditional RPC communication and therefore supports parameter type data.

The `use` attribute can be set to a value of “literal” or “encoded.” SOAP originally provided its own type system used to represent body content. Later, support for XML Schema data types was incorporated. This attribute value indicates which type system you want your message to use. The “literal” setting states that XML Schema data types will be applied.

When considering these two attributes, the four following combinations are possible and supported by SOAP:

- `style:RPC` + `use:encoded`
- `style:RPC` + `use:literal`
- `style:document` + `use:encoded`
- `style:document` + `use:literal`

The `style:document` + `use:literal` combination is preferred by SOA because it supports the notion of the document-style messaging model that is key to realizing the features of many WS-* specifications.

Case Study Example

In building the concrete part of the Employee service interface definition, TLS architects decide to use the `style:document` + `use:literal` combination, as shown in Example 8.10.

Example 8.10 The `binding` construct of the TLS Employee.wsdl document.

Click here to view code image

```
<binding name="EmployeeBinding"
  type="tns:EmployeeInterface">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetWeeklyHoursLimit">
    <soap:operation
      soapAction="http://www.xmltc.com/soapaction"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
```

```
    </operation>
    <operation name="UpdateHistory">
      <soap:operation
        soapAction="http://www.xmltc.com/soapaction"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
```

Chapter 9. Service API and Contract Design with REST Services and Microservices

Note

Parts of this chapter refer to HTTP syntax and REST-related

technologies that are covered in the *SOA with REST: Principles, Patterns & Constraints* series textbook.

REST service contracts are typically designed around the primary functions of HTTP methods, which make the documentation and expression of REST service contracts distinctly different from operation-based Web service contracts. Regardless of the differences in notation, the same overarching contract-first approach to designing REST service contracts is paramount when building services for a standardized service inventory.

With REST services in particular, the following benefits can be achieved:

- REST service contracts can be designed to logically group capabilities related to the functional contexts established during the service-oriented analysis process.
- Conventions can be applied to formally standardize resource names and input data representation.
- Complex methods can be defined and standardized to encapsulate a set of pre-defined interactions between a service and a service consumer.
- Service consumers are required to conform to the expression of the service contract, not vice versa.
- The design of business-centric resources and complex methods can be assisted by business analysts who may be able to help establish an accurate expression and behavior of business logic.

This chapter provides service contract design guidance for service candidates modeled as a result of the service-oriented analysis stage covered in Chapter 7.

Note that the physical design of REST service contract APIs may reveal functional requirements that are more suitable for alternative implementation mediums. The need to design a richer API or transactional functionality, for example, can warrant consideration of the use of SOAP-based Web services, as explained in Chapter 8.

SOA Patterns

As per the Dual Protocols [339] pattern, services within the same service inventory may be based on different implementation mediums and communication protocols. For example, REST services may reside alongside SOAP-based Web services.

The Concurrent Contracts [332] and Service Façade [360] patterns

can be further applied to enable the same body of service logic to expose alternative service contracts in support of two standard communication protocols.

9.1 Service Model Design Considerations

REST service contracts are based on the functional contexts established during the service-oriented analysis process. Depending on the nature of the functionality within a given context, each service will have already been categorized within a service model. Following are a set of service contract design considerations specific to each service model.

Entity Service Design

Each entity service establishes a functional boundary associated with one or more related business entities (such as invoice, claim, customer, and so on). The types of service capabilities exposed by a typical entity service are focused on functions that process the underlying data associated with the entity (or entities).

REST entity service contracts are typically dominated by service capabilities that include inherently idempotent and reliable GET, PUT, or DELETE methods. Entity services may need to support updating their state consistently with changes to other entity services. Entity services will also often include query capabilities for finding entities or parts of entities that match certain criteria, and therefore return hyperlinks to related and relevant entities.

If complex methods are permitted as part of a service inventory's design standards, then entity services may benefit from supplementing the standard HTTP method-based capabilities with the pre-defined interactions represented by complex methods.

Figure 9.1 provides an example of an entity service with two standard HTTP methods and two complex methods.

Figure 9.1 An entity service based on the Invoice business entity that defines a functional scope that limits the service capabilities to performing invoice-related processing. This agnostic Invoice service will be reused and composed by other services within the same service inventory in support of different automated business processes that need to process invoice-related data. This particular invoice service contract displays two service capabilities based on primitive methods and two service capabilities based on complex methods.

Complex methods are covered toward the end of this chapter in the *Complex Method Design* section.

SOA Patterns

The Entity Linking [342] pattern is commonly applied to REST-based entity services. As explained later in this chapter, REST services can process data represented by schemas, such as those provided by JSON and XML Schema specifications. With entity services in particular, this can place a great deal of emphasis on consistently applying the Canonical Schema [326] and Schema Centralization [356] patterns.

Utility Service Design

Like entity services, utility services are expected to be agnostic and reusable. However, unlike entity services, they do not usually have pre-defined functional scopes. Therefore, we need to confirm, before finalizing the service contract, that the method and resource combinations we've chosen during the service-oriented analysis phase are what we want to commit to for a given utility service design.

Whereas individual utility services group related service capabilities, the services' functional boundaries can vary dramatically. The example illustrated in Figure 9.2 is a utility service acting as a wrapper for a legacy system.

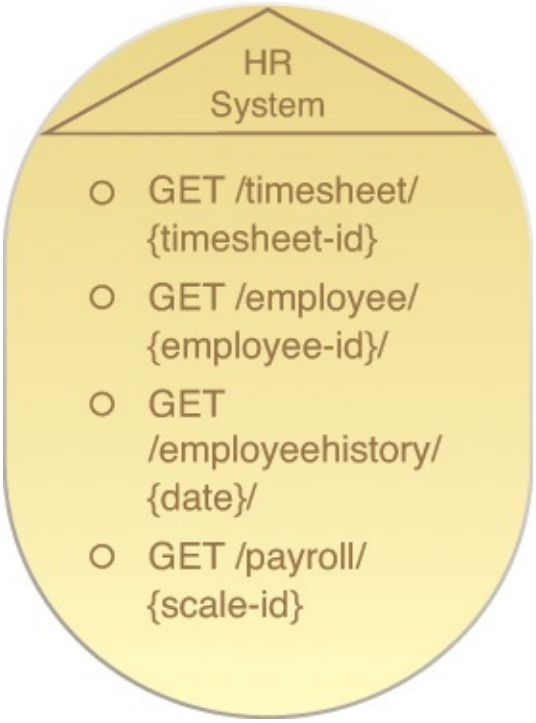


Figure 9.2 This utility service contract encapsulates a legacy HR system (and is accordingly named). The service capabilities it exposes provide generic, read-only data access functions against the data stored in the underlying legacy repository. For example, the Employee entity service (composed by the Verify Timesheet task service) may invoke an employee data-related service capability to retrieve data. This type of utility service may provide access to one of several available sources of employee and HR-related data.

SOA Patterns

Utility services are more likely to warrant support for alternative communication protocols, which makes the application of the Dual Protocols [339], Concurrent Contracts [332], and Service Façade

[360] patterns more likely than with entity services. Another pattern commonly applied during the utility service contract design stage is Legacy Wrapper [347].

Microservice Design

The predominant design consideration that applies to microservice contracts is the flexibility we have in how we can approach contract design. Due to the fact that microservices are typically based on an intentionally non-agnostic functional context, they will usually have limited service consumers. Sometimes a microservice may only have a single service consumer. Because we assume that the microservice will never need to facilitate any other service consumers in the future (because it is not considered reusable outside of a business process), the application of a number of service-orientation principles becomes optional.

Most notably, this includes the Standardized Service Contract (291) principle. Microservice APIs can be, to a certain extent, non-standard so that their individual capabilities can be optimized in support of their runtime performance and reliability requirements. This flexibility further carries over to the application of the Service Abstraction (294) and Service Loose Coupling (293) principles.

Exceptions to this design freedom pertain primarily to how the microservice interacts as part of the greater service composition. The cost of achieving the individual performance requirements of a microservice needs to be weighed against the requirements of the overall service-oriented solution it is a part of.

For example, the Standardized Service Contract (291) principle may need to be applied to an extent to ensure that a microservice contract is designed to support a standard schema that represents a common business document. Allowing the microservice to introduce a non-standard schema may benefit the processing efficiency of the microservice, but the resulting data transformation requirements for that data to be transformed into the standard schema used by the remaining service composition members may be unreasonable.

Figure 9.3 shows the service contract for the microservice that was modeled in Chapter 6.

Figure 9.3 A microservice contract with a single-purpose, non-agnostic functional scope. The service provides three capabilities specific to and in support of its parent business process.

SOA Patterns

In addition to the Dual Protocols [339], Concurrent Contracts [332], Service Façade [360], and Legacy Wrapper [347] patterns, REST-based microservices will commonly require the application of the Microservice Deployment [349] pattern and possibly the application of the Containerization [333] pattern.

It may be further required that artifacts to which a microservice may require access be replicated or redundantly deployed within the microservice implementation environment. These types of requirements can be addressed by implementation patterns such as Service Data Replication [358], Redundant Implementation [354], and even Composition Autonomy [331], if necessary.

Task Service Design

Task services will typically have few service capabilities, sometimes limited to only a single one. This is due to the fact that a task service contract's primary use is for the execution of automated business process (or task) logic. The service capability can be based on a simple verb, such as Start or Process. That verb, together with the name of the task service (that will indicate the nature of the task) is often all that is required for synchronous tasks. Additional service capabilities can be added to support asynchronous communication, such as

accessing state information or canceling an active workflow instance, as shown in Figure 9.4.

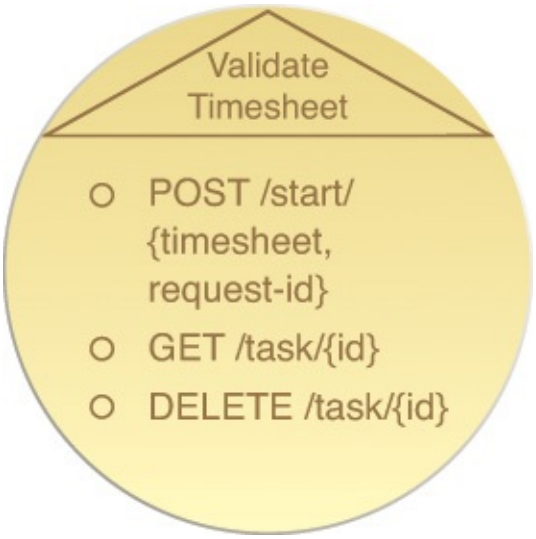


Figure 9.4 A sample task service, recognizable by the verb in its name. The contract only provides a service capability used by the composition initiator to trigger the execution of the Validate Timesheet business process that the task service logic encapsulates. In this case, the service capability receives a timesheet resource identifier that will be used as the basis of the validation logic, plus a unique consumer-generated request identifier that supports reliable triggering of the process. Two additional service capabilities allow consumers to asynchronously check on the progress of the timesheet validation task, and to cancel the task while it is in progress.

REST-based task services will often have service capabilities triggered by a POST request. However, this method is not inherently reliable. A number of techniques exist to achieve a reliable POST, including the inclusion of additional headers and handling of response messages, or the inclusion of a unique consumer-generated request identifier in the resource identifier.

To provide input to a parameterized task service it will make sense for the task service contract to include various identifiers into the capability's resource identifier template (that might have been parameters in a SOAP message). This frees up the service to expose additional resources rather than defining a custom media type as input to its processing.

If the task service automates a long-running business process it will return an interim response to its consumer while further processing steps may still need to take place. If the task service includes additional capabilities to check on or interact with the state of the business process (or composition instance), it will

typically include a hyperlink to one or more resources related to this state in the initial response message.

Case Study Example

MUA follows proven REST service contract design techniques together with custom design standards established specifically for the MUA enterprise. Architects use select service candidates modeled in Chapter 7 as the basis for their service contract designs.

Confer Student Award Service Contract (Task)

A student who submits an award conferral application will do so through a Web browser. A separate user interface is therefore designed to allow users to enter the application details. It is the submission of this browser-based form that initiates the task service.

Upon receiving the submission, a server-side script organizes the form data into an XML document based on the following media type:

Click here to view code image

```
application/vnd.edu.mua.student-award-conferral-
application+xhtml+xml
```

Example 9.1 provides a submitted application form completed with sample data collected from the human user. This represents the data set that kickstarts and drives the execution of an entire instance of the Confer Student Award business process.

Example 9.1 Sample application data, as submitted to the Web server. This document structure contains both human-readable and machine-processable information.

Click here to view code image

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" >
  <head>
    <title>Student Award Conferral Application</title>
  </head>
  <body>
    <p>Student:
      <a rel="student"
```

```
        href="http://student.mua.edu/student/555333">
        John Smith (Student Number 555333)
      </a>
    </p>
    <p>Award:
      <a rel="award"
        href="http://award.mua.edu/awardBS/CompSci">
        Bachelor of Science with Computer Science Major
      </a>
    </p>
    <p>Event:
      <a rel="event"
        href="http://event.mua.edu/achievement">
        Outstanding Achievement
      </a>
    </p>
  </body>
</html>
```

Figure 9.5 displays the Confer Student Award service contract. The preceding media type is deliberately designed to include human-readable and machine-readable data in a form suitable for long-term archival. The document is submitted to a service capability corresponding directly to the Start capability defined in the Confer Student Award service candidate.

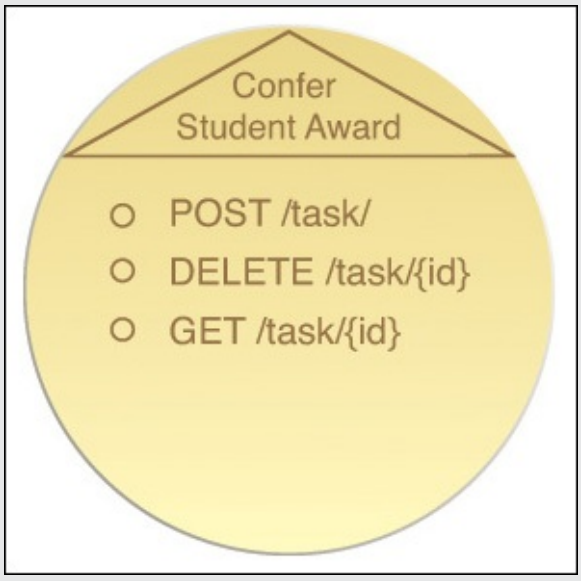


Figure 9.5 The Confer Student Award service contract.

As also shown in Figure 9.5, during the design process for this service contract it was decided to add new service capabilities to provide the following functions:

- *DELETE task{id}* – This capability was added to allow an

executing instance of the Confer Student Award business process to be terminated.

- *GET task{id}* – This capability allows the state of an executing instance of the Confer Student Award business process to be queried.

Note that the sensitive nature of this kind of application means that the GET *task*{id} capability can be accessed only by authorized staff and by the student. The DELETE *task*{id} capability is only accessible by the student to cancel the application process.

Event Service Contract (Entity)

The Event entity service is equipped with a GET *event*{id} service capability, which is used to query event information and corresponds to the Get Details capability candidate from the Event service candidate (Figure 9.6).

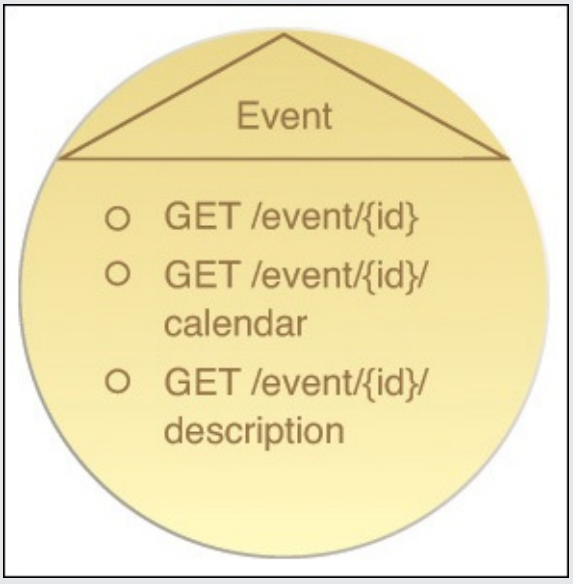


Figure 9.6 The Event service contract.

During the service-oriented design process, architects decided to add further GET *event*{id}/calendar and GET *event*{id}/description capabilities that allow for the retrieval of more specific event information. These capabilities were not added specifically in support of the Confer Student Award business process, but more so to provide a broader range of anticipated reusable functionality.

Award Service Contract (Entity)

In addition to implementing the three service capabilities from the original Award service candidate (Figure 9.6), some of MUA's SOA architects decide to make some further changes.

Back in Chapter 7, MUA analysts determined that the following action was to be encompassed by the Confer Student Award task service logic:

- Verify Student Transcript Qualifies for Award Based on Award Conferral Rules

However, with the rules being specific to each award type they determine that it should be the Award entity service that applies the bulk of these rules. Nevertheless, some generic checks do need to be applied so the logic is divided between the Confer Student Award task service and the Award entity service.

To avoid requiring the task service to pass full transcript details into the Award entity service for verification, it is decided to use a code-on-demand approach. The Award entity service provides the logic, but the logic is executed by the task service. The decision to define the logic centrally within the Award entity service is justified based on the need to produce human-readable output (for students), alongside machine-readable output (for the Confer Student Award service). As a result, the Entity service provides a new GET *award*conferral-rules service capability (Figure 9.7) that supports the output of two formats for the rules logic: the first in human-readable form and the second in a form that can be readily embedded into the task service's logic.

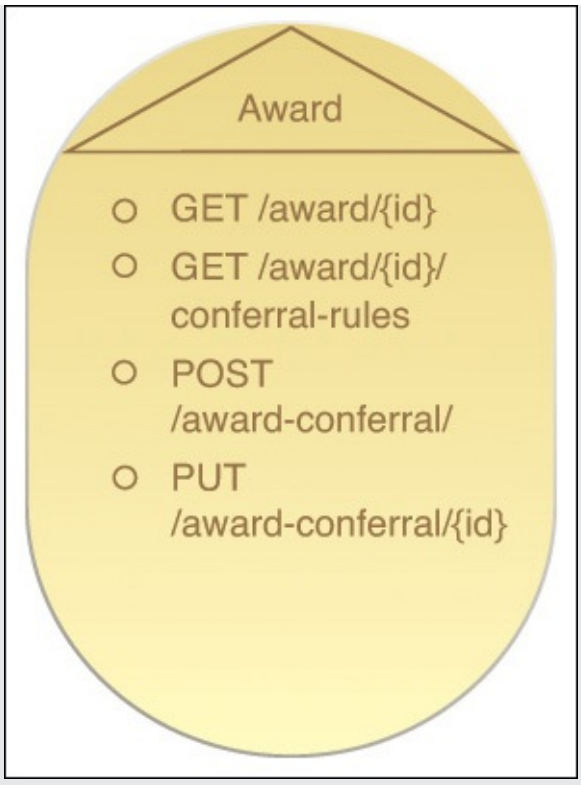


Figure 9.7 The Award service contract.

MUA architects choose JavaScript for this purpose because they find that JavaScript runtimes are readily available for many of the technology platforms that have been used to develop services within the inventory. Choosing JavaScript over other technologies also accounts for its being the language of choice for the user interface tier of the service inventory.

The same service capability is able to return the conferral rules in JavaScript or as human-readable HTML. The decision as to which transformation to carry out depends on which `Accept` header was provided by the service consumer. For example, the Confer Student Award task service requests the `application/javascript` media type, while service consumers requiring human-readable output will request the `text/html` media type.

Student Transcript Service Contract (Entity)

The Student service was originally intended as a centralized entity service that would encompass all student-related functionality and data access. However, iterations of the REST service modeling process that occurred subsequent to the examples covered in

Chapter 7 resulted in a service inventory blueprint that revealed the Student service candidate as being far more coarsely grained than any other. This was primarily due to the complexity of the Student entity and its relationships to other related entities.

Upon review of the Student service candidate, it was determined to create a set of student-related entity services. One of these more specialized variations became the Student Transcript service candidate (Figure 9.8).

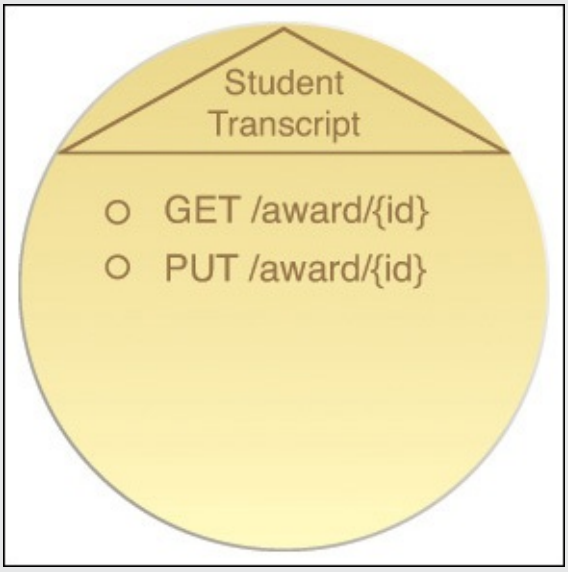


Figure 9.8 The Student Transcript service candidate that was defined subsequent to the Student service candidate. This service effectively replaces the Student service in the Confer Student Award service composition.

Because the Confer Student Award business process only requires access to student transcript information, it only needs to compose the Student Transcript service, not the actual Student service. As shown in Figure 9.9, the Student Transcript service contains service capabilities that correspond to the service capability candidates provided by the Student Transcript service candidate.

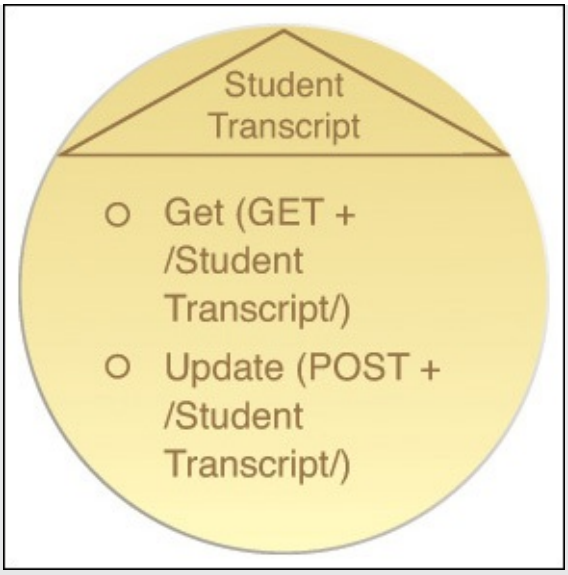


Figure 9.9 The Student Transcript service contract.

Notification and Document Service Contracts (Utility)

The Notification service and Document service process similar human-readable data. Notifications sent via email or hard copy can both be encoded as a human-readable document format, such as HTML or PDF.

The Notification service is retained for email notifications while the Document service has been evolved into a printer-centric and postal delivery–centric utility service. The Confer Student Award task service can send a document to the student in the preferred format by looking up the preferred delivery method in the original application form.

As shown in Figure 9.10, the Notification and Document services can each be invoked with the POST method.

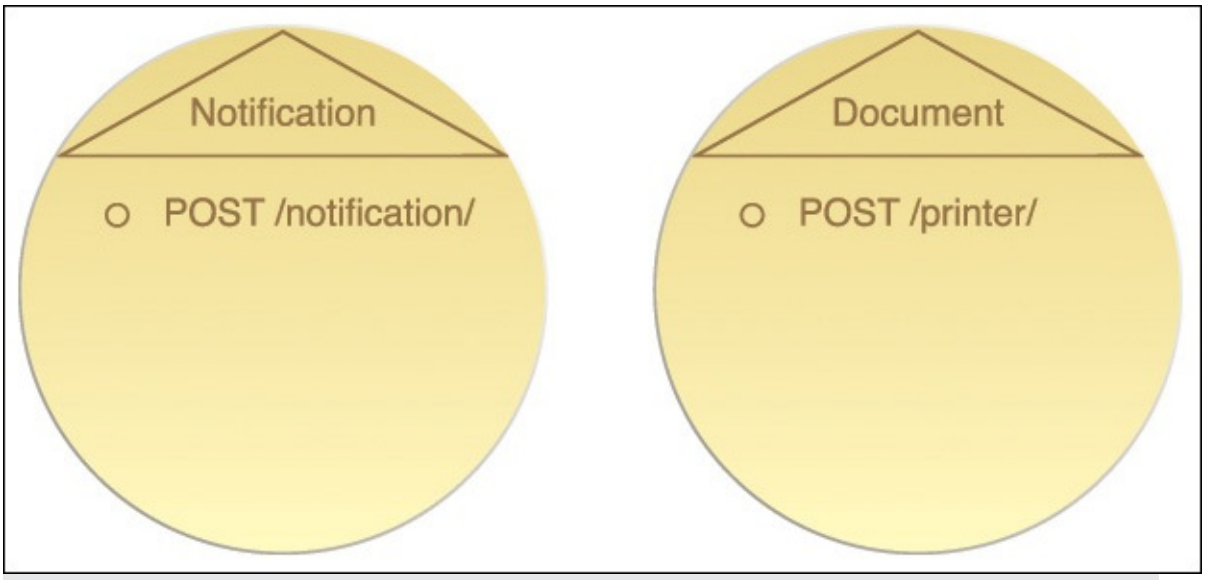


Figure 9.10 The Notification and Document service contracts.

The sample student (John Smith) from the application form used as input for the Confer Student Award task service has nominated his contact preference with a hyperlink to `mailto:s555333@student.mua.edu`. The service inventory standard for handling such an address is to transform the URL into `http://notification.mua.edu/sender?to=s555333@student.mua.edu` and use a POST method for its delivery. John Smith's notification will be delivered via email to this address.

9.2 REST Service Design Guidelines

The following is a series of common guidelines and considerations for designing REST service contracts.

Uniform Contract Design Considerations

When creating a uniform contract for a service inventory, we have a responsibility to equip and limit its features so that it is streamlined to effectively accommodate requirements and restrictions unique to the service inventory. The default characteristics of Web-centric technology architecture can provide an effective basis for a service inventory's uniform contract, although additional forms of standardization and customization are likely required for non-trivial service inventory architectures.

The following sections explore how common elements of a uniform contract

(methods, media types, and exceptions in particular) can be customized to meet the needs of individual service inventories.

Designing and Standardizing Methods

When we discuss methods in relation to the uniform contract, it is considered shorthand for a request-response communications mechanism that also includes methods, headers, response codes, and exceptions. Methods are centralized as part of the uniform contract to ensure that there are always a small number of ways of moving information around within a particular service inventory, and that existing service consumers will work correctly with new or modified services as they are added to the inventory. Although it is important to minimize the number of methods in the uniform contract, methods can and should be added when service inventory interaction requirements demand it. This is a natural part of evolving a service inventory in response to business change.

Note

Less well-known HTTP methods have come and gone in the past. For example, at various times the HTTP specification has included a PATCH method consistent with a partial update or partial store communications mechanism. PATCH is currently specified separately from HTTP methods in the IETF's RFC 5789 document. Other IETF specifications, such as WebDAV's RFC 4918 and the Session Initiation Protocol's RFC 3261, introduced new methods as well as new headers and response codes (or special interpretations thereof).

HTTP provides a solid foundation by supplying the basic set of methods (such as GET, PUT, DELETE, POST) proven by use on the Web and widely supported by off-the-shelf software components and hardware devices. But the need to express other types of interactions for a service inventory may arise. For example, you may decide to add a special method that can be used to reliably trigger a resource to execute a task at most once, rather than using the less reliable HTTP POST method.

HTTP is designed to be extended in these ways. The HTTP specification explicitly supports the notion of extension methods, customized headers, and extensibility in other areas. Leveraging this feature of HTTP can be effective, as long as new extensions are added carefully and at a rate appropriate for the number of services that implement HTTP within an inventory. This way, the total

number of options for moving data around (that services and consumers are required to understand) remains manageable.

Note

At the end of this chapter we explore a set of sample extended methods (referred to as complex methods). Each utilizes multiple basic HTTP methods or a single basic HTTP method multiple times to perform pre-defined, standardized message interactions.

Common circumstances that can warrant the creation of new methods include:

- Hyperlinks may be used to facilitate a sequence of request-response pairs. When they start to read like verbs instead of nouns and tend to suggest that only a single method will be valid on the target of a hyperlink, we can consider introducing a new method instead. For example, the “customer” hyperlink for an invoice resource suggests that GET and PUT requests might be equally valid for the customer resource. But a “begin transaction” hyperlink or a “subscribe” hyperlink suggests only POST is valid and may indicate the need for a new method instead.
- Data with must-understand semantics may be needed within message headers. In this case, a service that ignores this metadata can cause incorrect runtime behavior. HTTP does not include a facility for identifying individual headers or information within headers as “must-understand.” A new method can be used to enforce this requirement because the custom method will be automatically rejected by a service that doesn’t understand the request (whereas falling back on a default HTTP method will allow the service to ignore the new header information).

It is important to acknowledge that introducing custom methods can have negative impacts when exploring vendor diversity within an implementation environment. It may prevent off-the-shelf components (such as caches, load balancers, firewalls, and various HTTP-based software frameworks) from being fully functional within the service inventory. Stepping away from HTTP and its default methods should only be attempted in mature service inventories when the effects on the underlying technology architecture and infrastructure are fully understood.

Some alternatives to creating new methods can also be explored. For example, service interactions that require a number of steps can use hyperlinks to guide consumers through the requests they need to make. The HTTP Link header (RFC

5988) can be considered to keep these hyperlinks separate from the actual document content.

SOA Patterns

Working with and customizing the uniform interface pertains to the natural application of the Reusable Contract [355] pattern.

Designing and Standardizing HTTP Headers

Exchanging messages with metadata is common in service-oriented solution design. Because of the emphasis on composing a set of services together to collectively automate a given task at runtime, there is often a need for a message to provide a range of header information that pertains to how the message should be processed by intermediary service agents and services along its message path.

Built-in HTTP headers can be used in a number of ways:

- To add parameters related to a request method as an alternative to using query strings to represent the parameters within the URL. For example, the `Accept` header can supplement the GET method by providing content negotiation data.
- To add parameters related to a response code. For example, the `Location` header can be used with the `201 Created` response code to indicate the identifier of a newly created resource.
- To communicate general information about the service or consumer. For example, the `Upgrade` header can indicate that a service consumer supports and prefers a different protocol, while the `Referrer` header can indicate which resource the consumer came from while following a series of hyperlinks.

This type of general metadata may be used in conjunction with any HTTP method.

HTTP headers can also be utilized to add rich metadata. For this purpose custom headers are generally required, which reintroduces the need to determine whether or not the message content must be understood by recipients or whether it can optionally be ignored. This association of must-understand semantics with new methods and must-ignore semantics with new message headers is not an inherent feature of REST, but it is a feature of HTTP.

When introducing custom HTTP headers that can be ignored by services, regular HTTP methods can safely be used. This also makes the use of custom headers

backwards-compatible when creating new versions of existing message types.

As previously stated in the *Designing and Standardizing Methods* section, new HTTP methods can be introduced to enforce must-understand content by requiring services to either be designed to support the custom method or to reject the method invocation attempt altogether. In support of this behavior, a new `Must-Understand` header can be created in the same format as the existing `Connection` header, which would list all the headers that need to be understood by message recipients.

If this type of modification is made to HTTP, it would be the responsibility of the SOA Governance Program Office responsible for the service inventory to ensure that these semantics are implemented consistently as part of inventory-wide design standards. If custom, must-understand HTTP headers are successfully established within a service inventory, we can explore a range of applications of messaging metadata. For example, we can determine whether it is possible or feasible to emulate messaging metadata such as what is commonly used in SOAP messaging frameworks based on WS-* standards.

While custom headers that enforce reliability or routing content (as per the WS-ReliableMessaging and WS-Addressing standards) can be added to recreate acknowledgement and intelligent load balancing interactions, other forms of WS-* functions are subject to built-in limitations of the HTTP protocol. The most prominent example is the use of WS-Security to enable message-level security features, such as encryption and digital signatures. Message-level security protects messages by actually transforming the content so that intermediaries along a message path are unable to read or alter message content. Only those message recipients with prior authorization are able to access the content.

This type of message transformation is not supported in HTTP/1.1. HTTP does have some basic features for transforming the body of the message alone through its `Content-Encoding` header, but this is generally limited to compression of the message body and does not include the transformation of headers. If this feature was used for encryption purposes the meaning of the message could still be modified or inspected in transit, even though the body part of the message could be protected. Message signatures are also not possible in HTTP/1.1 as there is no canonical form for an HTTP message to sign, and no industry standard that determines what modifications intermediaries would be allowed to make to such a message.

Designing and Standardizing HTTP Response Codes

HTTP was originally designed as a synchronous, client-server protocol for the exchange of HTML pages over the World Wide Web. These characteristics are compatible with REST constraints and make it also suitable as a protocol used to invoke REST service capabilities.

Developing a service using HTTP is very similar to publishing dynamic content on a Web server. Each HTTP request invokes a REST service capability and that invocation concludes with the sending of a response message back to the service consumer.

A given response message can contain any one of a wide variety of HTTP codes, each of which has a designated number. Certain ranges of code numbers are associated with particular types of conditions, as follows:

- `100-199` are informational codes used as low-level signaling mechanisms, such as a confirmation of a request to change protocols.
- `200-299` are general success codes used to describe various kinds of success conditions.
- `300-399` are redirection codes used to request that the consumer retry a request to a different resource identifier, or via a different intermediary.
- `400-499` represent consumer-side error codes that indicate that the consumer has produced a request that is invalid for some reason.
- `500-599` represent service-side error codes that indicate that the consumer's request may have been valid but that the service has been unable to process it for internal reasons.

The consumer-side and service-side exception categories are helpful for "assigning blame" but do little to actually enable service consumers to recover from failure. This is because, while the codes and reasons provided by HTTP are standardized, how service consumers are required to behave upon receiving response codes is not. When standardizing service design for a service inventory, it is necessary to establish a set of conventions that assign response codes concrete meaning and treatment.

Table 9.1 provides common descriptions of how service consumers can be designed to respond to common response codes.

Response Code	Reason Phrase	Treatment
100	Continue	Indeterminate
101	Switching Protocols	Indeterminate
1xx	Any other 1xx code	Failure
200	OK	Success
201	Created	Success
202	Accepted	Success
203	Non-Authoritative Information	Success
204	No Content	Success
205	Reset Content	Success
206	Partial Content	Success
2xx	Any other 2xx code	Success
300	Multiple Choices	Failure

301	Moved Permanently	Indeterminate (Common Behavior: Modify resource identifier and retry.)
302	Found	Indeterminate (Common Behavior: Change request to a GET and retry using nominated resource identifier.)
303	See Other	
304	Not Modified	Success (Common Behavior: Return cached response to consumer.)
305	Use Proxy	Indeterminate (Common Behavior: Connect to identified proxy and resend original message.)
307	Temporary Redirect	Indeterminate (Common Behavior: Retry once to nominated resource identifier.)
3xx	Any other 3xx code	Failure
400	Bad Request	Failure

401	Unauthorized	Indeterminate (Common Behavior: Retry with correct credentials.)
402	Payment Required	Failure
403	Forbidden	Failure
404	Not Found	Success if request was DELETE, else Failure
405	Method Not Allowed	Failure
406	Not Acceptable	Failure
407	Proxy Authentication Required	Indeterminate (Common Behavior: Retry with correct credentials.)
408	Request Timeout	Failure
409	Conflict	Failure
410	Gone	Success if request was DELETE, else Failure

411	Length Required	Failure
412	Precondition Failed	Failure
413	Request Entity Too Large	Failure
414	Request-URI Too Long	Failure
415	Unsupported Media Type	Failure
416	Requested Range Not Satisfiable	Failure
417	Expectation Failed	Failure
4xx	Any other 4xx code	Failure
500	Internal Server Error	Failure
501	Not Implemented	Failure
502	Bad Gateway	Failure
503	Service Unavailable	Repeat if Retry-After header is specified. Otherwise, Failure.
504	Gateway Timeout	Repeat if request is idempotent. Otherwise, Failure.
505	HTTP Version Not Supported	Failure
5xx	Any other 5xx code	Failure

Table 9.1 HTTP response codes and typical corresponding consumer behavior.

As is evident when reviewing Table 9.1, HTTP response codes go well beyond the simple distinction between success and failure. They provide an indication of how consumers can respond to and recover from exceptions.

Let's take a closer look at some of the values from the Treatment column in Table 9.1:

- *Repeat* means that the consumer is encouraged to repeat the request, taking into account any delay specified in responses such as `503`

`Service Unavailable`. This may mean sleeping before trying again. If the consumer chooses not to repeat the request, it must treat the method as failed.

- *Success* means the consumer should treat the message transmission as a successful action and must therefore not repeat it. (Note that specific success codes may require more subtle interpretation.)
- *Failed* means that the consumer must not repeat the request unchanged, although it may issue a new request that takes the response into account. The consumer should treat this as a failed method if a new request cannot be generated. (Note that specific failure codes may require more subtle interpretation.)
- *Indeterminate* means that the consumer needs to modify its request in the manner indicated. The request must not be repeated unchanged and a new request that takes the response into account should be generated. The final outcome of the interaction will depend on the new request. If the consumer is unable to generate a new request then this code must be treated as failed.

Because HTTP is a protocol and not a set of message processing logic, it is up to the service to decide what status code (success, failure, or otherwise) to return. As previously mentioned, because consumer behavior is not always sufficiently standardized by REST for machine-to-machine interactions, it needs to be explicitly and meaningfully standardized as part of an SOA project.

For example, indeterminate codes tend to indicate that service consumers must handle a situation using their own custom logic. We can standardize these types of codes in two ways:

- Design standards can determine which indeterminate codes can and cannot be issued by service logic.
- Design standards can determine how service consumer logic must interpret those indeterminate codes that are allowed.

Customizing Response Codes

The HTTP specification allows for extensions to response codes. This extension feature is primarily there to allow future versions of HTTP to introduce new codes. It is also used by some other specifications (such as WebDAV) to define custom codes. This is typically done with numbers that are not likely to collide with new HTTP codes, which can be achieved by putting them near the end of the particular range (for example, 299 is unlikely to ever be used by the main HTTP standard).

Specific service inventories can follow this approach by introducing custom response codes as part of the service inventory design standards. In support of the Uniform Contract {311} constraint, custom response codes should only be defined at the uniform contract level, not at the REST service contract level.

When creating custom response codes, it is important that they be numbered based on the range they fall in. For example, 2xx codes should be communicating success, while 4xx codes should only represent failure conditions.

Additionally, it is good practice to standardize the insertion of human-readable content into the HTTP response message via the Reason Phrase. For example, the code 400 has a default reason phrase of "Bad Request." This is enough for a service consumer to handle the response as a general failure, but it doesn't tell a human anything useful about the actual problem. Setting the reason phrase to "The service consumer request is missing the Customer address field" or perhaps even "Request body failed validation against schema http://example.com/customer" is more helpful, especially when reviewing logs of exception conditions that may not have the full document attached.

Consumers can associate generic logic to handle response codes in each of these ranges, but may also need to associate specific logic to specific codes. Some codes can be limited so that they are only generated if the consumer requests a special feature of HTTP, which means that some codes can be left unimplemented by consumers that do not request these features.

Uniform contract exceptions are generally standardized within the context of a particular new type of interaction that is required between services and consumers. They will typically be introduced along with one or more new methods and/or headers. This context will guide the kind of exceptions that are created. For example, it may be necessary to introduce a new response code to indicate that a request cannot be fulfilled due to a lock on a resource. (WebDAV provides the `423 Locked` code for this purpose.)

When introducing and standardizing custom response codes for a service inventory uniform contract, we need to ensure that:

- Each custom code is appropriate and absolutely necessary
- The custom code is generic and highly reusable by services
- The extent to which service consumer behavior is regulated and is not too restrictive so that the code can apply to a large range of potential situations
- Code values are set to avoid potential collision with response codes from relevant external protocol specifications

- Code values are set to avoid collision with custom codes from other service inventories (in support of potential cross-service inventory message exchanges that may be required)

Response code numeric ranges can be considered a form of exception inheritance. Any code within a particular range is expected to be handled by a default set of logic, just as if the range were the parent type for each exception within that range.

In this section we have briefly explored response codes within the context of HTTP. However, it is worth noting that REST can be applied with other protocols (and other response codes). It is ultimately the base protocol of a service inventory architecture that will determine how normal and exceptional conditions are reported.

For example, you could consider having a REST-based service inventory standardized on the use of SOAP messages that result in SOAP-based exceptions instead of HTTP exception codes. This allows the response code ranges to be substituted for inheritance of exceptions.

Designing Media Types

During the lifetime of a service inventory architecture we can expect more changes will be required to the set of a uniform contract's media types than to its methods. For example, a new media type will be required whenever a service or consumer needs to communicate machine-readable information that does not match the format or schema requirements of any existing media type.

Some common media types from the Web to consider for service inventories and service contracts include:

- `text/plain; charset=utf-8` for simple representations, such as integer and string data. Primitive data can be encoded as strings, as per built-in XML Schema data types
- `application/xhtml+xml` for more complex lists, tables, human-readable text, hypermedia links with explicit relationship types, and additional data based on microformats.org and other specifications
- `application/json` for a lightweight alternative to XML that has broad support by programming languages
- `text/uri-list` for plain lists of URIs
- `application/atom+xml` for feeds of human-readable event information or other data collections that are time-related (or time ordered)

Before inventing new media types for use within a service inventory, it is advisable to first carry out a search of established industry media types that may be suitable.

Whether choosing existing media types or creating custom ones, it is helpful to consider the following best practices:

- Each specific media type should ideally be specific to a schema. For example, `application/xml` or `application/json` are not schema-specific, while `application/atom+xml` used as a syndication format is specific enough to be useful for content negotiation and to identify how to process documents.
- Media types should be abstract in that they specify only as much information as their recipients need to extract via their schemas. Keeping media types abstract allows them to be reused within more service contracts.
- New media types should reuse mature vocabularies and concepts from industry specifications whenever appropriate. This reduces the risk that key concepts have been missed or poorly constructed, and further improves compatibility with other applications of the same vocabularies.
- A media type should include a hyperlink whenever it needs to refer to a related resource whose representation is located outside the immediate document. Link relation types may be defined by the media type's schema or, in some cases, separately, as part of a link relation profile.
- Custom media types should be defined with must-ignore semantics or other extension points that allow new data to be added to future versions of the media type without old services and consumers rejecting the new version.
- Media types should be defined with standard processing instructions that describe how a new processor should handle old documents that may be missing some information. Usually these processing instructions ensure that earlier versions of a document have compatible semantics. This way, new services and consumers do not have to reject the old versions.

All media types that are either invented for a particular service inventory or reused from another source should be documented in the uniform contract profile, alongside the definition of uniform methods.

HTTP uses Internet media type identifiers that conform to a specific syntax. Custom media types are usually identified with the notation:

Click here to view code image

```
application/vnd.organization.type+supertype
```

where `application` is a common prefix that indicates that the type is used for machine consumption and standards. The `organization` field identifies the vendor namespace, which can optionally be registered with IANA.

The `type` part is a unique name for the media type within the organization, while the `supertype` indicates that this type is a refinement of another media type. For example, `application/vnd.com.examplebooks.purchase-order+xml` may indicate that:

- The type is meant for machine consumption.
- The type is vendor-specific, and the organization that has defined the type is "examplebooks.com."
- The type is for purchase orders (and may be associated with a canonical Purchase Order XML schema).
- The type is derived from XML, meaning that recipients can unambiguously handle the content with XML parsers.

Types meant for more general interorganizational use can be defined with the media type namespace of the organization ultimately responsible for defining the type. Alternatively, they can be defined without the vendor identification information in place by registering each type directly, following the process defined in the RFC 4288 specification.

SOA Patterns

The Content Negotiation [334] pattern formalizes the native ability of REST services to process media type information at runtime.

Designing Schemas for Media Types

Within a service inventory, most custom media types created to represent business data and documents will be defined using XML Schema or JSON Schema. This can essentially establish a set of standardized data models that are reused by REST services within the inventory to whatever extent feasible.

For this to be successful, especially with larger collections of services, schemas need to be designed to be flexible. This means that it is generally preferable for schemas to enforce a coarse level of validation constraint granularity that allows

each schema to be applicable for use with a broader range of data interaction requirements.

REST requires that media types and their schemas be defined only at the uniform contract level. If a service capability requires a unique data structure for a response message, it must still use one of the canonical media types provided by the uniform contract. Designing schemas to be flexible and weakly typed can accommodate a variety of service-specific message exchange requirements, but perhaps not for all cases.

Example 9.2 provides an example of a flexible schema design.

Example 9.2 One of the most straightforward ways of making a media type more reusable is to design the schema to support a list of zero or more items. This enables the media type to permit one instance of the underlying type, but also allows queries that return zero or more instances. Making individual elements within the document optional can also increase reuse potential.

Click here to view code image

```
Media type = application/vnd.com.actioncon.po+xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://example.org/schema/po"
  xmlns="http://example.org/schema/po">
  <xsd:element name="LineItemList" type="LineItemListType"/>
  <xsd:complexType name="LineItemListType">
    <xsd:element name="LineItem" type="LineItemType"
      minOccurs="0"/>
  </xsd:complexType>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:anyURI"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

SOA Patterns

The Validation Abstraction [365] pattern provides a technique for intentionally weakly typing XML Schema definitions (which is also explored in Chapters 6, 12, and 13 in the book titled *Web Service Contract Design and Versioning for SOA*). The Content Negotiation [334] pattern can be applied to address the option of having a single

REST service support two alternative schemas.

It is technically possible for individual REST service contracts to introduce contract-specific XML schemas, but in doing so we need to accept that the Uniform Contract {311} constraint will be violated.

This may be warranted when a service capability needs to generate a response message containing unique data (or a unique combination of data) for which:

- No suitable canonical schemas exist
- No new canonical schema should be created due to the fact that it would not be reusable by other services

A consequence of non-compliance to Uniform Contract {311} is potentially increased levels of negative coupling between service consumers and the service offering service capabilities based on service-specific media types. Service-specific media types should be clearly identified and effort should be made to minimize the quantity of logic that is directly exposed to and made dependent upon these types.

Complex Method Design

The uniform contract establishes a set of base methods used to perform basic data communication functions. As we've explained, this high-level of functional abstraction is what makes the uniform contract reusable to the extent that we can position it as the sole, overarching data exchange mechanism for an entire inventory of services. Besides its inherent simplicity, this part of a service inventory architecture automatically results in the baseline standardization of service contract elements and message exchange.

The standardization of HTTP on the World Wide Web results in a protocol specification that describes the things that services and consumers "may," "should," or "must" do to be compliant with the protocol. The resulting level of standardization is intentionally only as high as it needs to be to ensure the basic functioning of the Web. It leaves a number of decisions as to how to respond to different conditions up to the logic within individual services and consumers. This "primitive" level of standardization is important to the Web, where we can have numerous foreign service consumers interacting with third-party services at any given time.

A service inventory, however, often represents an environment that is private and controlled within an IT enterprise. This gives us the opportunity to customize this standardization beyond the use of common and primitive methods. This form of customization can be justified when we have requirements for increasing the levels of predictability and quality-of-service beyond what the World Wide Web can provide.

For example, let's say that we would like to introduce a design standard whereby all accounting-related documents (invoices, purchase orders, credit notes, etc.) must be retrieved with logic that, upon encountering a retrieval failure, automatically retries the retrieval a number of times. The logic would further require that subsequent retrieval attempts do not alter the state of the resource representing the business documents (regardless of whether a given attempt is successful).

With this type of design standard, we are essentially introducing a set of rules and requirements as to how the retrieval of a specific type of document needs to be carried out. These are rules and requirements that cannot be expressed or enforced via the base, primitive methods provided by HTTP. Instead, we can apply them in addition to the level of standardization enforced by HTTP by assembling them (together with other possible types of runtime functions) into aggregate interactions. This is the basis of the *complex method*.

A complex method encapsulates a pre-defined set of interactions between a service and a service consumer. These interactions can include the invocation of standard HTTP methods. To better distinguish these base methods from the complex methods that encapsulate them, we'll refer to base HTTP methods as *primitive methods* (a term only used when discussing complex method design).

Complex methods are qualified as "complex" because they:

- Can involve the composition of multiple primitive methods
- Can involve the composition of a primitive method multiple times
- Can introduce additional functionality beyond method invocation
- Can require optional headers or properties to be supported by or included in messages

As previously stated, complex methods are generally customized for and standardized within a given service inventory. For a complex method to be standardized, it needs to be documented as part of the service inventory architecture specification. We can define a number of common complex methods as part of a uniform contract that then become available for implementation by all services within the service inventory.

Complex methods have distinct names. The complex method examples that we're covering are called:

- *Fetch* – A series of GET requests that can recover from various exceptions
- *Store* – A series of PUT or DELETE requests that can recover from various exceptions
- *Delta* – A series of GET requests that keep a consumer in sync with changing resource state
- *Async* – An initial modified request and subsequent interactions that support asynchronous request message processing

Services that support a complex method communicate this by showing the method name as part of a separate service capability (Figure 9.11), alongside the primitive methods that the complex method is built upon. When project teams create consumer programs for certain services, they can determine the required consumer-side logic for a complex method by identifying what complex methods the service supports, as indicated by its published service contract.

Figure 9.11 An Invoice service contract displaying two service capabilities based on primitive methods and two service capabilities based on complex methods. We can initially assume that the two complex methods incorporate the use of the two primitive methods, and proceed to confirm this by studying the design specification that documents the complex methods.

Note

When applying the Service Abstraction (294) principle to REST service composition design, we may exclude entirely describing some of the primitive methods from the service contract. This can be the result of design standards that only allow the use of a complex method in certain situations. Going back to the previous example about the use of a complex method for retrieving accounting-related documents, we may have a design standard that prohibits these documents from being retrieved via the regular GET method (because the GET method does not enforce the additional reliability requirements).

It is important to note that the use of complex methods is by no means required. Outside of controlled environments in which complex methods can be safely defined, standardized, and applied in support of the Increased Intrinsic Interoperability goal, their use is uncommon and generally not recommended. When building a service inventory architecture, we can opt to standardize on certain interactions through the use of complex methods or we can choose to limit REST service interaction to the use of primitive methods only. This decision will be based heavily on the distinct nature of the business requirements addressed and automated by the services in the service inventory.

Despite their name, complex methods are intended to add simplicity to service inventory architecture. For example, let's imagine we decide not to use pre-defined complex methods and then realize that there are common rules or policies that we applied to numerous services and their consumers. In this case, we will have built the common interaction logic redundantly across each individual consumer-service pair. Because the logic was not standardized, its redundant implementations will likely exist differently. When we need to change the common rules or policies, we will need to revisit each redundant implementation accordingly. This maintenance burden and the fact that the implementations will continue to remain out of sync make this a convoluted architecture that is unnecessarily complex. This is exactly the problem that the use of complex methods is intended to avoid.

The upcoming sections introduce a set of sample complex methods organized into two sections:

- *Stateless Complex Methods*
- *Stateful Complex Methods*

Note that these methods are by no means industry standard. Their names and the type of message interactions and primitive method invocations they encompass have been customized to address common types of functionality.

Note

The *Case Study Example* at the end of this chapter further explores this subject matter. In this example, in response to specific business requirements, two new complex methods (one stateless, the other stateful) are defined.

Stateless Complex Methods

This first collection of complex methods encapsulates message interactions that are compliant with the Stateless {308} constraint.

Fetch Method

Instead of relying only on a single invocation of the HTTP GET method (and its associated headers and behavior) to retrieve content, we can build a more sophisticated data retrieval method with features such as

- Automatic retry on timeout or connection failure
- Required support for runtime content negotiation to ensure the service consumer receives data in a form it understands
- Required redirection support to ensure that changes to the service contract can be gracefully accommodated by service consumers
- Required cache control directive support by services to ensure minimum latency, minimum bandwidth usage, and minimum processing for redundant requests

We'll refer to this type of enhanced read-only complex method as a Fetch. Figure 9.12 shows an example of a pre-defined message interaction of a Fetch method designed to perform content negotiation and automatic retries.

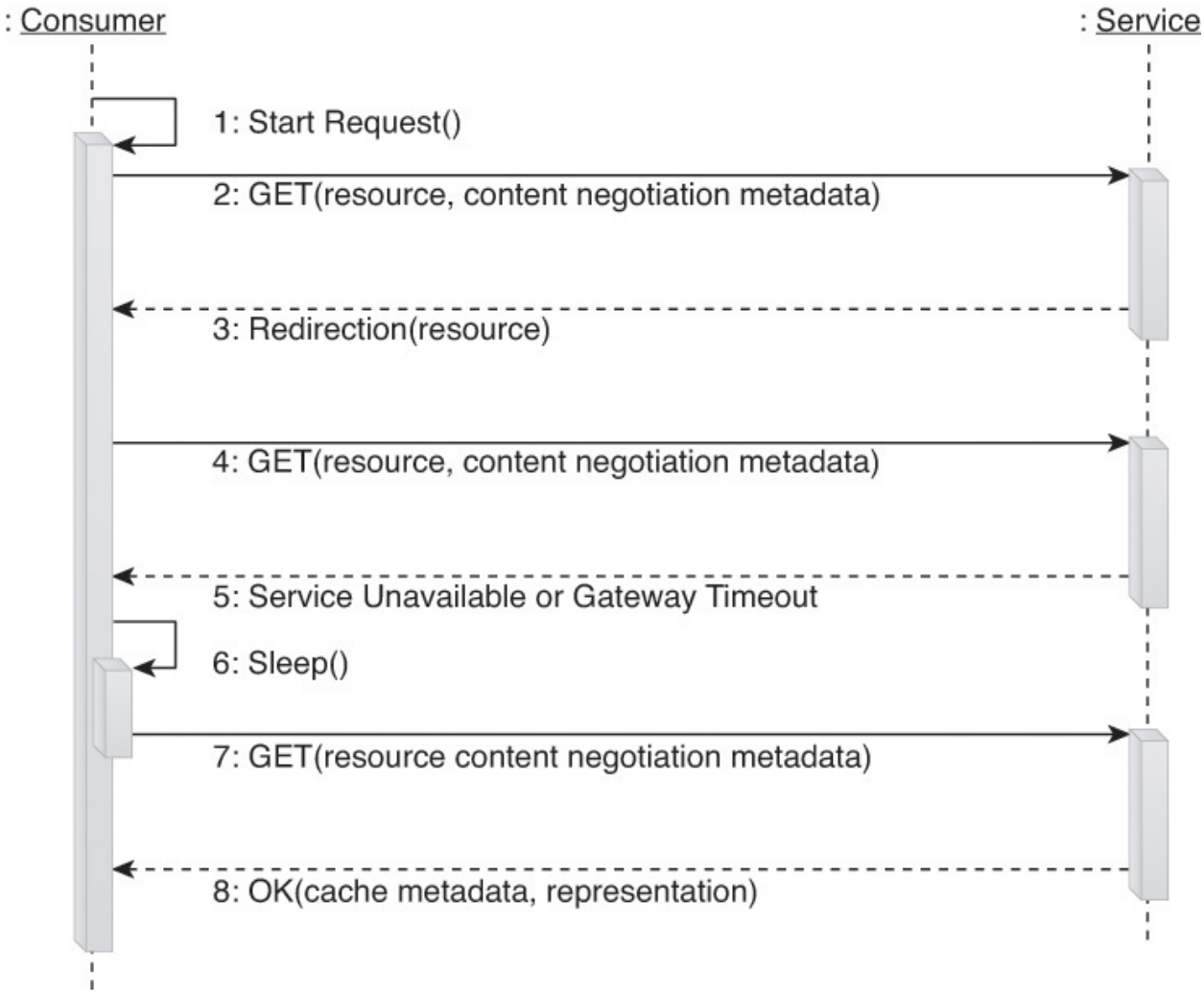


Figure 9.12 An example of a Fetch complex method comprised of consecutive GET method calls.

Store Method

When using the standard PUT or DELETE methods to add new resources, set the state of existing resources, or remove old resources, service consumers can suffer request timeouts or exception responses. Although the HTTP specification explains what each exception means, it does not impose restrictions as to how they should be handled. For this purpose, we can create a custom Store method to standardize necessary behavior.

The Store method can have a number of the same features as a Fetch, such as requiring automatic retry of requests, content negotiation support, and support for redirection exceptions. Using PUT and DELETE, it can also defeat low bandwidth connections by always sending the most recent state requested by the consumer, rather than needing to complete earlier requests first.

The same way that individual primitive HTTP methods can be idempotent, the Store method can be designed to behave idempotently. By building upon primitive idempotent methods, any repeated, successful request messages will have no further effect after the first request message is successfully executed.

For example, when setting an invoice state from “Unpaid” to “Paid”:

- A “toggle” request would not be idempotent because repeating the request toggles the state back to “Unpaid.”
- The “PUT” request is idempotent when setting the invoice to “Paid” because it has the same effect, no matter how many times the request is repeated.

It is important to understand that the Store and its underlying PUT and DELETE requests are requests *to* service logic, not an action carried out on the service’s underlying database. As shown in Figure 9.13, these types of requests are stated in an idempotent manner in order to efficiently allow for the retrying of requests without the need for sequence numbers to add reliable messaging support.

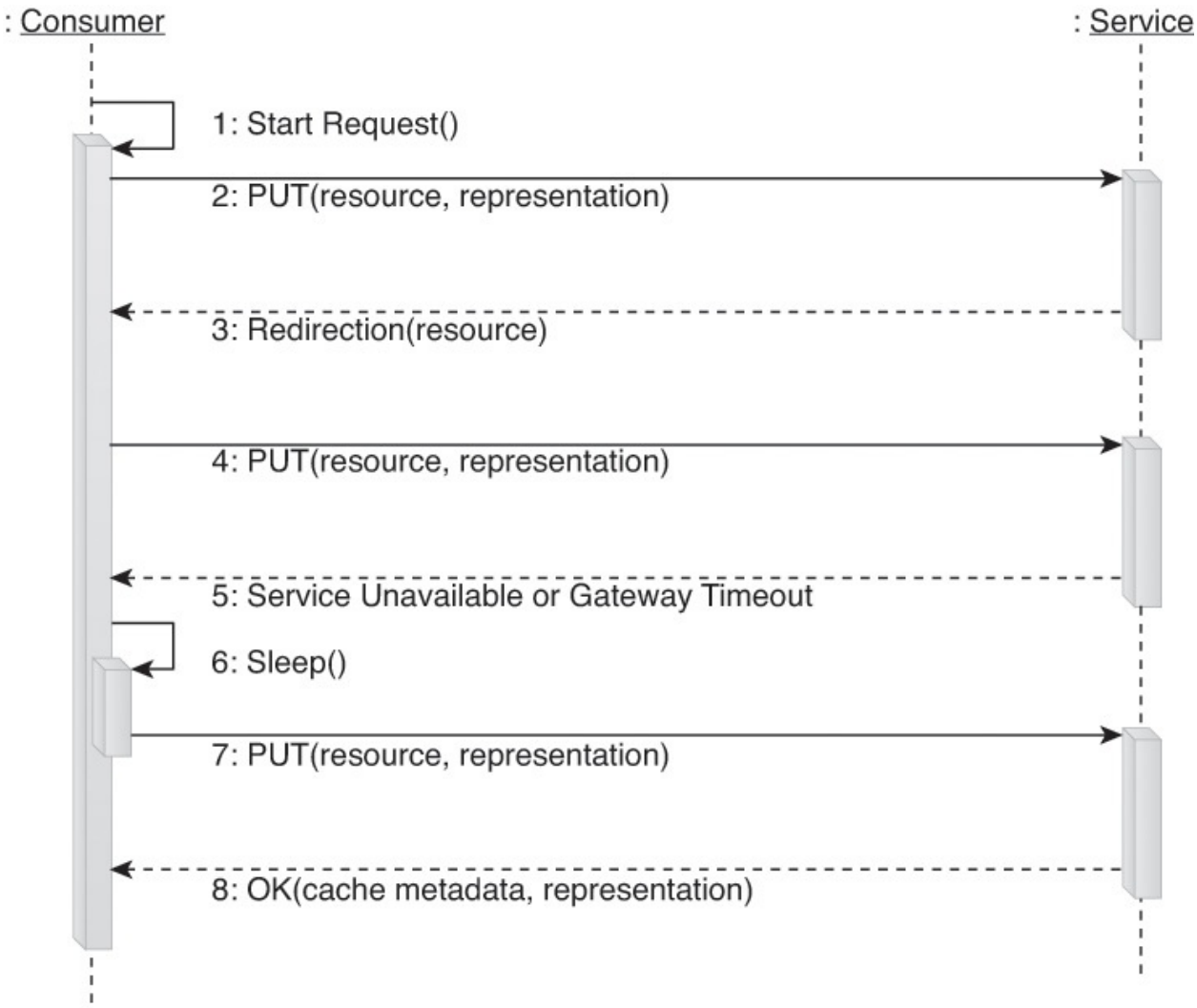


Figure 9.13 An example of the interaction carried out by a Store complex method.

Note

Service capabilities that incorporate this type of method are an example of the application of the Idempotent Capability [345] pattern.

Delta Method

It is often necessary for a service consumer to remain synchronized with the state of a changing resource. The Delta method is a synchronization mechanism that facilitates stateless synchronization of the state of a changing resource between the service that owns this state and consumers that need to stay in alignment with the state.

The Delta method follows processing logic based on the following three basic functions:

1. The service keeps a history of changes to a resource.
2. The consumer gets a URL referring to the location in the history that represents the last time the consumer queried the state of the resource.
3. The next time the consumer queries the resource state, the service (using the URL provided by the consumer) returns a list of changes that have occurred since the last time the consumer queried the resource state.

Figure 9.14 illustrates this using a series of GET invocations.

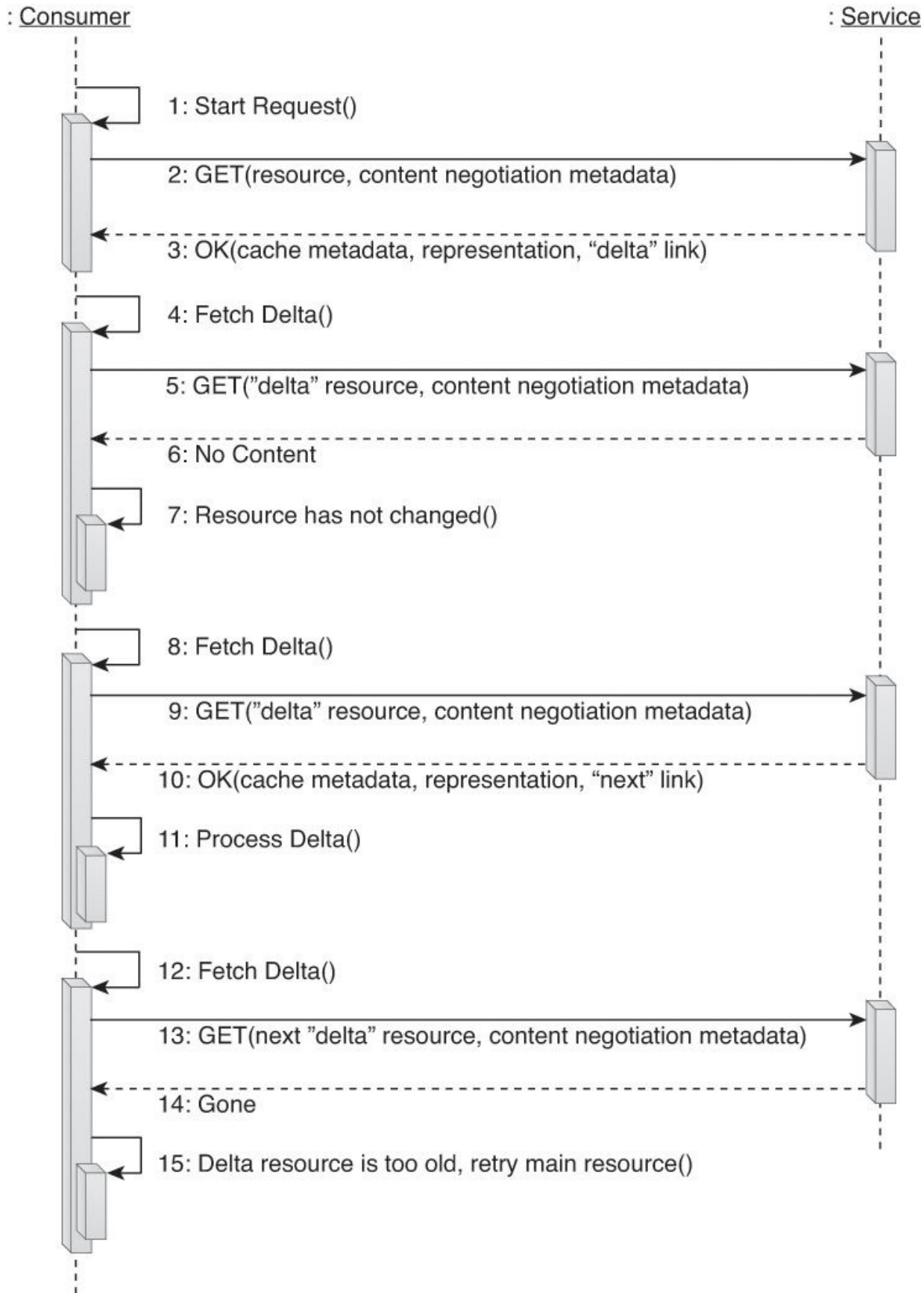

: Consumer
: Service
1: Start Request()
2: GET(resource, content negotiation metadata)
3: OK(cache metadata, representation, “delta” link)
4: Fetch Delta()
5: GET(”delta” resource, content negotiation metadata)
6: No Content
7: Resource has not changed()
8: Fetch Delta()
9: GET(”delta” resource, content negotiation metadata)
10: OK(cache metadata, representation, “next” link)
11: Process Delta()
12: Fetch Delta()
13: GET(next ”delta” resource, content negotiation metadata)
14: Gone
15: Delta resource is too old, retry main resource()

Figure 9.14 An example of the message interaction encompassed by the Delta complex method.

The service provides a “main” resource that responds to GET requests by returning the current state of the resource. Next to the main resource it provides a collection of “delta” resources that each return the list of changes from a nominated point in the history buffer.

The consumer of the Delta method activates periodically or when requested by the core consumer logic. If it has a delta resource identifier it sends its request to that location. If it does not have a delta resource identifier, it retrieves the main resource to become synchronized. In the corresponding response the consumer receives a link to the delta for the current point in the history buffer. This link will be found in the `Link` header (RFC 5988) with relation type `Delta`.

The requested delta resource can be in any one of the following states:

1. It can represent a set of one or more changes that have occurred to the main resource since the point in history that the delta resource identifier refers to. In this case, all changes in the history from the nominated point are returned along with a link to the new delta for the current point in the history buffer. This link will be found in the `Link` header with relation type `Next`.
2. It may not have a set of changes because no changes have occurred since its nominated point in the history buffer, in which case it can return the `204 No Content` response code to indicate that the service consumer is already up-to-date and can continue using the delta resource for its next retrieval.
3. Changes may have occurred, but the delta has already expired because the nominated point in history is now so old that the service has elected not to preserve the changes. In this situation, the resource can return a `410 Gone` code to indicate that the consumer has lost synchronization and should re-retrieve the main resource.

Delta resources use the same caching strategy as the main resource.

The service controls how many historical deltas it is prepared to accumulate, based on how much time it expects consumers will take (on average) to get up-to-date. In certain cases where a full audit trail is maintained for other purposes, the number of deltas can be indefinite. The amount of space required to keep this record is constant and predictable regardless of the number of consumers, leaving each individual service consumer to keep track of where it is in the

history buffer.

Async Method

This complex method provides pre-defined interactions for the successful and canceled exchange of asynchronous messages. It is useful for when a given request requires more time to execute than what the standard HTTP request timeouts allow.

Normally if a request takes too long, the consumer message processing logic will time out or an intermediary will return a `504 Gateway Timeout` response code to the service consumer. The Async method provides a fallback mechanism for handling requests and returning responses that does not require the service consumer to maintain its HTTP connection open for the total duration of the request interaction.

As shown in Figure 9.15, the service consumer issues a request, but does so specifying a callback resource identifier. If the service chooses to use this identifier, it responds with the `202 Accepted` response code, and may optionally return a resource identifier in the `Location` header to help it track the place of the asynchronous request in its processing queue.

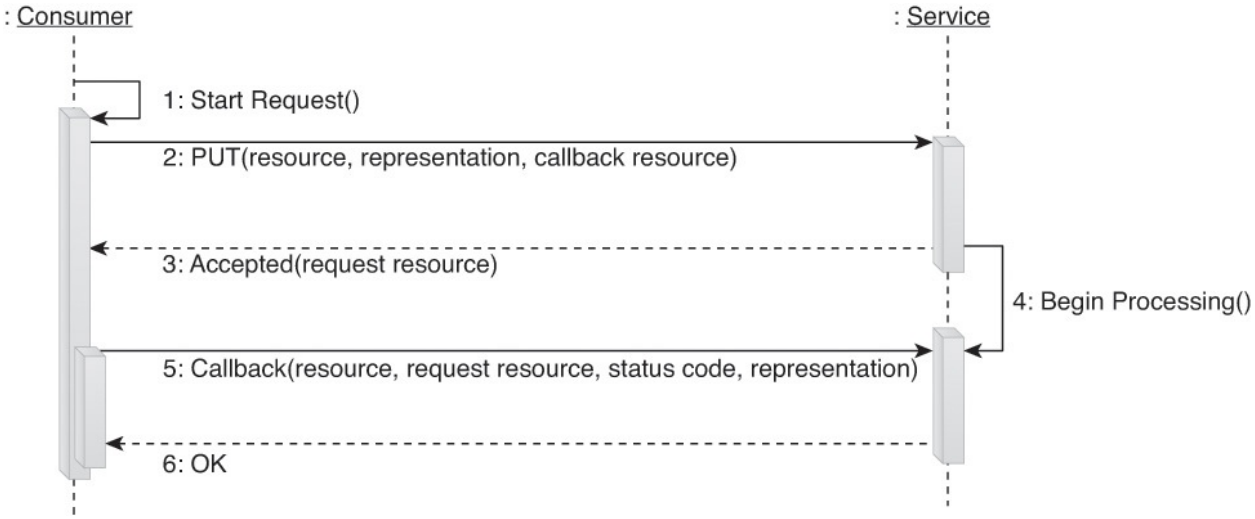


Figure 9.15 An asynchronous request interaction encompassed by the Async complex method.

When the request has been fully processed, its result is delivered by the service, which then issues a request to the callback address of the service consumer. If the service consumer issues a DELETE request (as shown in Figure 9.16) while the Async request is still in the processing queue (and before a response is returned), a separate pre-defined interaction is carried out to cancel the

asynchronous request. In this case, no response is returned and the service cancels the processing of the request.

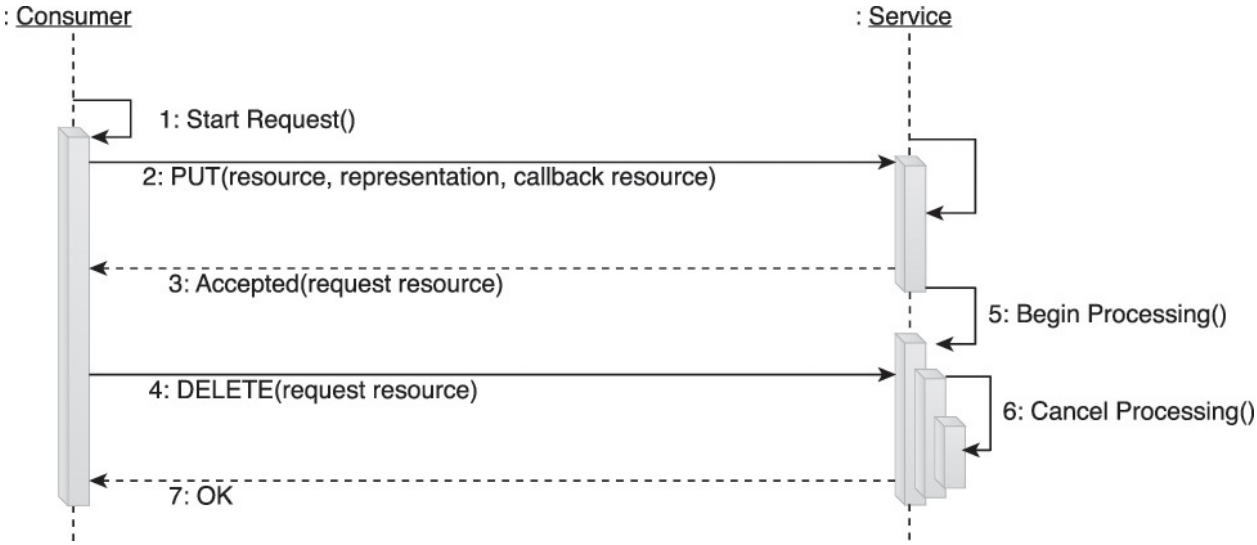


Figure 9.16 An asynchronous cancel interaction encompassed by the Async complex method.

If the consumer cannot listen for callback requests, it can use the asynchronous request identifier to periodically poll the service. After the request has been successfully handled, it is possible to retrieve its result using the previously described Fetch method before deleting the asynchronous request state. Services that execute either interaction encompassed by this method must have a means of purging old asynchronous requests if service consumers are unavailable to pick up responses or otherwise "forget" to delete request resources.

Stateful Complex Methods

The following two complex methods use REST as the basis of service design but incorporate interactions that intentionally breach the Stateless {308} constraint. Although the scenarios represented by these methods are relatively common in traditional enterprise application designs, this kind of communication is not considered native to the World Wide Web. The use of stateful complex methods can be warranted when we accept the reduction in scalability that comes with this design decision.

Trans Method

The Trans method essentially provides the interactions necessary to carry out a two-phase commit between one service consumer and one or more services. Changes made within the transaction are guaranteed to either successfully

propagate across all participating services, or all services are rolled back to their original states.

This type of complex method requires a “prepare” function for each participant before a final commit or rollback is carried out. Functionality of this sort is not natively supported by HTTP. Therefore, we need to introduce a custom PREP-PUT method (a variant of the PUT method), as shown in Figure 9.17.

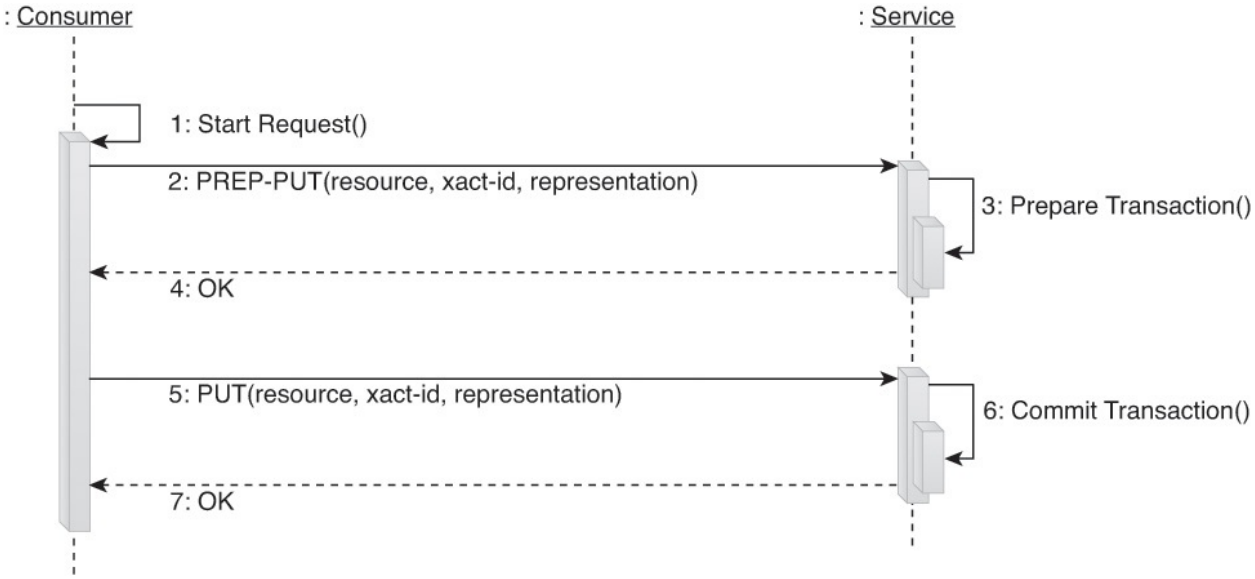


Figure 9.17 An example of a Trans complex method, using a custom primitive method called PREP-PUT.

In this example the PREP-PUT method is the equivalent of PUT, but it does not commit the PUT action. A different method name is used to ensure that if the service does not understand how to participate in the Trans complex method, it then rejects the PREP-PUT method and allows the consumer to abort the transaction.

Carrying out the logic behind a typical Trans complex method will usually require the involvement of a transaction controller to ensure that the commit and rollback functions are truly and reliably carried out with atomicity.

PubSub Method

A variety of publish-subscribe options are available after it is decided to intentionally breach the Stateless {308} constraint. These types of mechanisms are designed to support real-time interactions in which a service consumer must act immediately when some pre-determined event at a given resource occurs.

There are various ways that this complex method can be designed. Figure 9.18 illustrates an approach that treats publish-subscribe messaging as a “cache-

invalidation” mechanism.

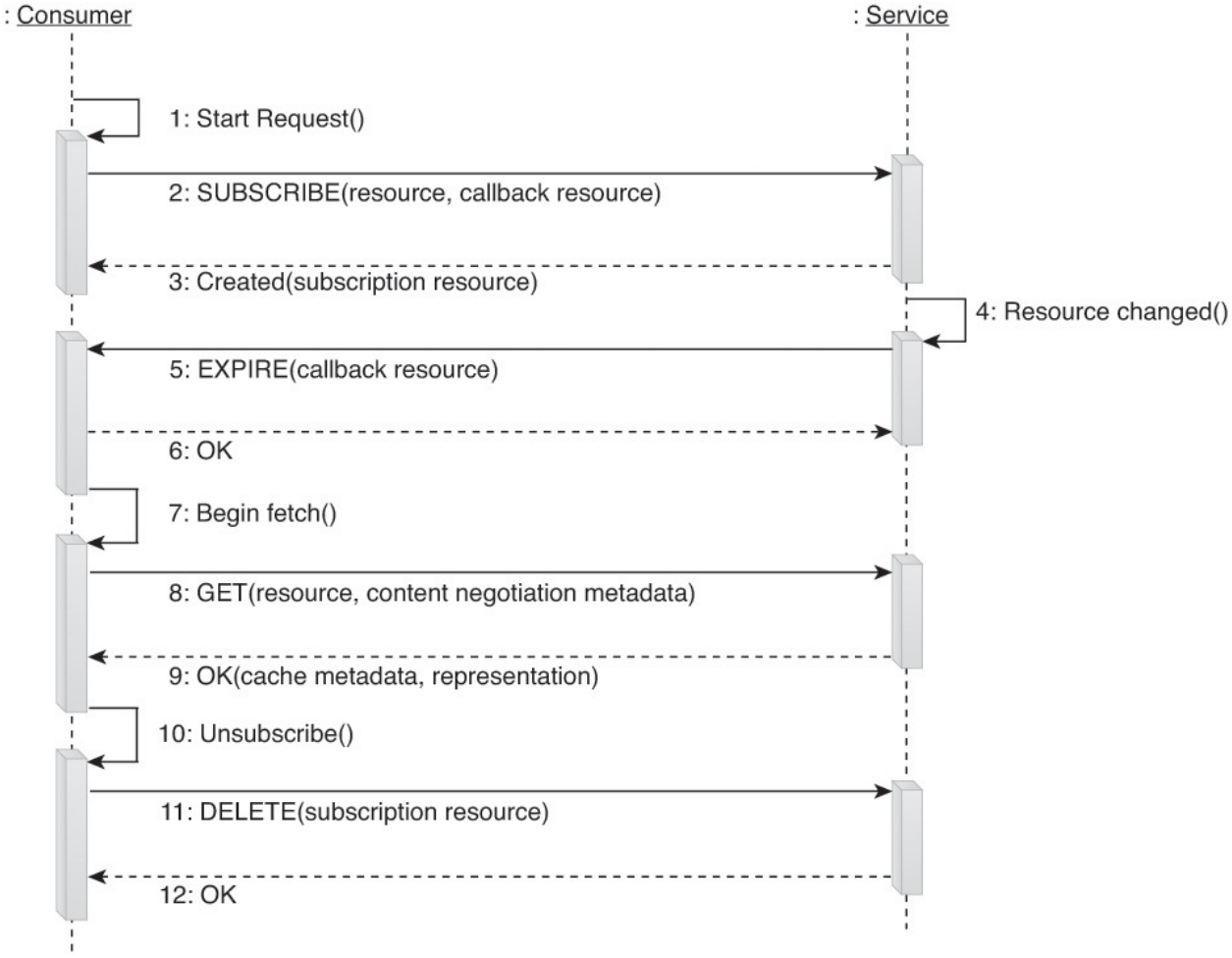


Figure 9.18 An example of a PubSub complex method based on cache invalidation. When the service determines that something has changed on one or more resources, it issues cache expiry notifications to its subscribers. Each subscriber can then use a Fetch complex method (or something equivalent) to bring the subscriber up-to-date with respect to the changes.

This form of publish-subscribe interaction is considered “lightweight” because it does not require services to send out the actual changes to the subscribers. Instead, it informs them that a resource has changed by pushing out the resource identifier, and then reuses an existing, cacheable Fetch method as the service consumers pull the new representations of the changed resource.

The amount of state required to manage these subscriptions is bound to one fixed-sized record for each service consumer. If multiple invalidations queue up for a particular subscribed event, they can be folded together into a single notification. Regardless of whether the consumer receives one or multiple invalidation messages, it will still only need to invoke one Fetch method to bring

itself up-to-date with the state of its resources each time it sees one or more new invalidation messages.

The PubSub method can be further adjusted to distribute subscription load and session state storage to different places around the network. This technique can be particularly effective within cloud-based environments that naturally provide multiple, distributed storage resources.

SOA Patterns

The Event-Driven Messaging [343] pattern can be applied in support of this complex method. It provides an alternative to the repeated polling of the resource, which can negatively impact performance if the polling frequency is increased to detect changes with minimal delay.

Case Study Example

The MUA team responsible for service design encounters a number of requirements for accessing and updating resource state. For example:

- One service consumer needs to atomically read the state of the resource, perform processing, and store the updated state back to the resource.
- Another service consumer needs to support concurrent user actions that modify the same resource. These actions update certain resource properties while others need to remain the same.

Allowing individual services consumers to contain different custom logic that performs these types of functions will inadvertently lead to problems and runtime exceptions when any two service consumers attempt updates to the same resource at the same time.

MUA architects conclude that the simplest way to avoid this is to introduce a new complex method that ensures that a resource is locked while being updated by a given consumer. Using the rules of optimistic locking, an approach commonly used with database updates, they are able to create a complex method that is stateless and takes advantage of existing standard features of the HTTP protocol. They name the method “OptLock” and write up an official description that is made part of the uniform contract profile.

OptLock Complex Method

If two separate service consumers attempt to update the state of a resource at the same time, their actions will clearly conflict with each other as the outcome depends on the order in which their requests reach the service. The OptLock method (Figure 9.19) addresses this problem by providing a means by which a service consumer can determine whether the state of a resource has changed since it was last read by the consumer before attempting an update.

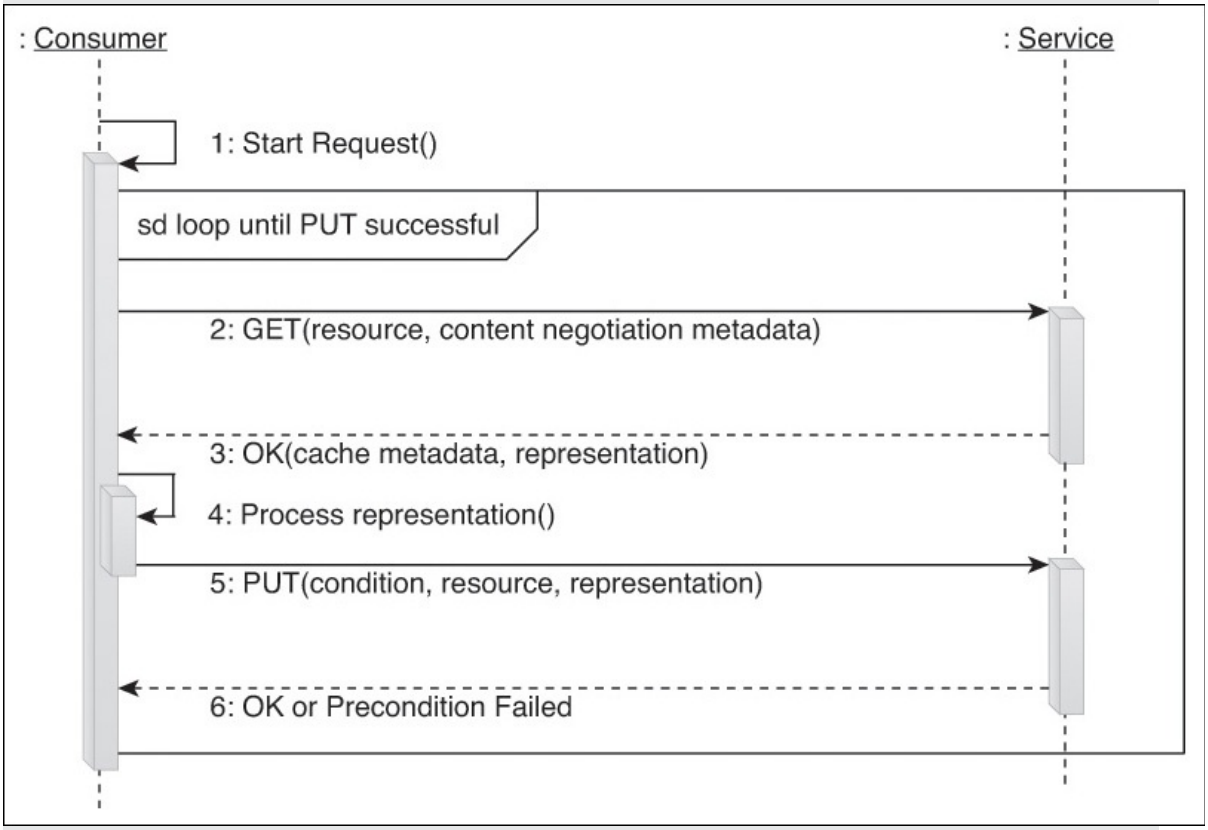


Figure 9.19 An example of an OptLock complex method.

Specifically, a consumer will first retrieve the current state associated with a resource identifier using the Fetch method. Along with the data, the consumer also receives an "ETag." ETag is a concept from HTTP that uniquely identifies the version of a resource in an opaque fashion. Whenever the resource changes state, its ETag is guaranteed to be different. When the service consumer initiates a Store, it does so conditionally by requesting the service to only honor the Store interaction if the resource's ETag still matches the one that it had when fetched. This is done with the `If-Match` header. The service can use the ETag value in the

condition to detect whether the resource state has been changed in the meantime.

The OptLock complex method does not introduce any new features to HTTP, but instead introduces new requirements for the handling of GET and PUT requests. Specifically, the GET request must return an ETag value and the PUT request must process the `If-Match` header. Additionally, if the resource has changed, the service must further guarantee not to carry out the PUT request.

There are several techniques for computing ETags. Some compute a hash value out of the state information associated with the resource, some simply keep a “last modified” timestamp for each resource, and others track the version of the resource state explicitly.

The OptLock method may not scale effectively for high concurrent access to a particular resource. If consumer update requests are denied with an HTTP `409 Conflict` response code, the OptLock method prescribes how the consumer can recover by fetching a newer version of the resource over which they have to recompute the change and retry the Store method. However, this may fail again due to a conflicting update request. Service consumers that interact with a resource in this way rely on that particular resource having relatively low rates of write access.

The OptLock complex method becomes available as part of the uniform contract and is implemented by several services. However, scenarios emerge where multiple consumers attempt to modify the resource at the same time, causing regular exceptions and failed updates. These situations occur during peak usage times, and because concurrent usage volume is expected to increase further, it is determined that a more efficient means of serializing updates to the resource needs to be established.

It is proposed that the OptLock complex method be changed to perform pessimistic locking instead, as per the following PesLock complex method description.

PesLock Complex Method

Pessimistic locking provides greater flexibility and certainty than optimistic locking. From a REST perspective, this comes at the cost of introducing stateful interactions and limiting concurrent access

while the pessimistic lock is held.

As shown in Figure 9.20, the WebDAV extensions to HTTP provide locking primitives that can be used within a composition architecture that intentionally breaches the Stateless {308} constraint. One consumer may lock out others from accessing a resource, so care must be taken that appropriate access control policies are in place. Consumers can also fail while the lock is held, which means that locks must be able to time out independently of the consumers that register them.

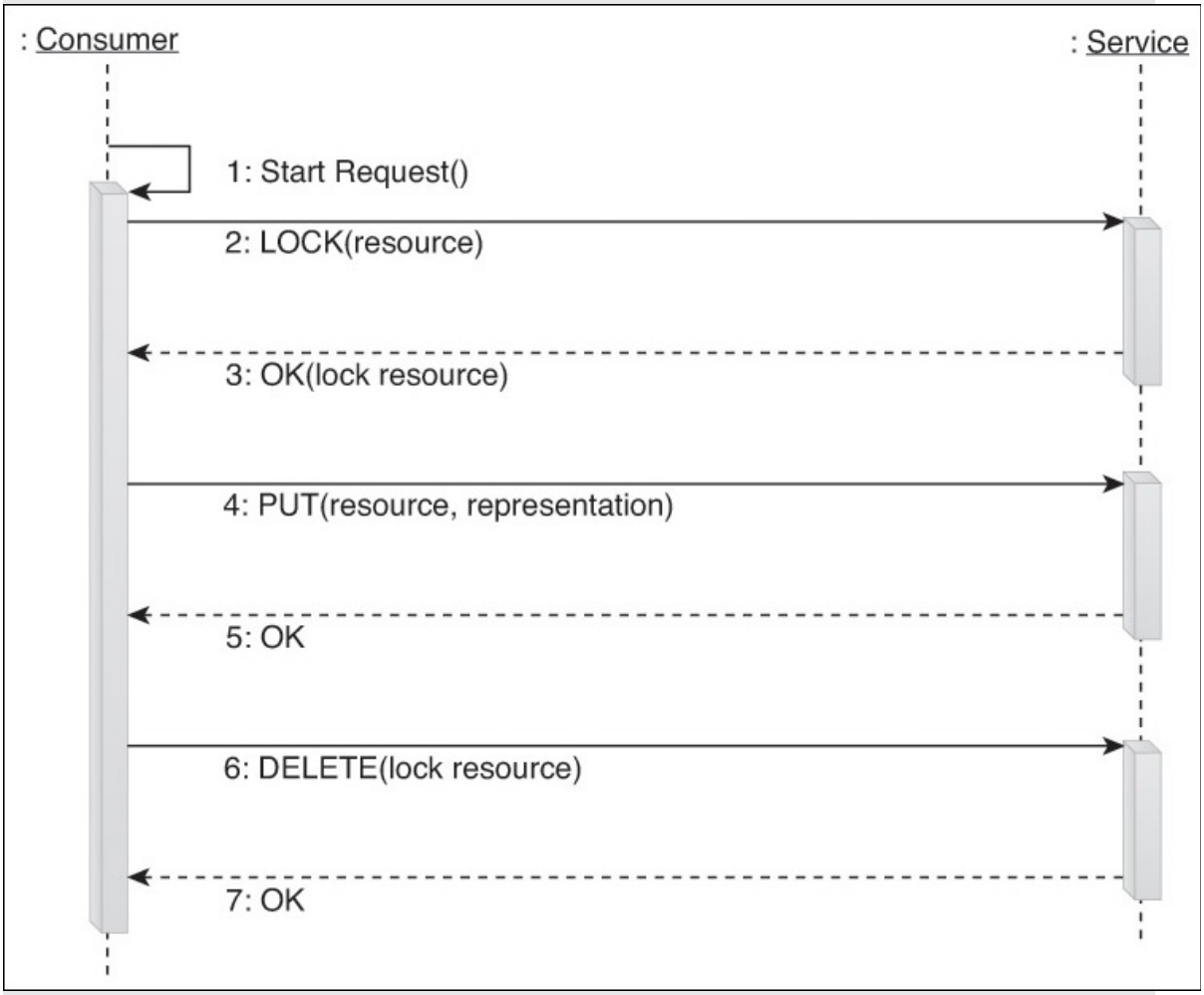


Figure 9.20 An example of a PesLock complex method.

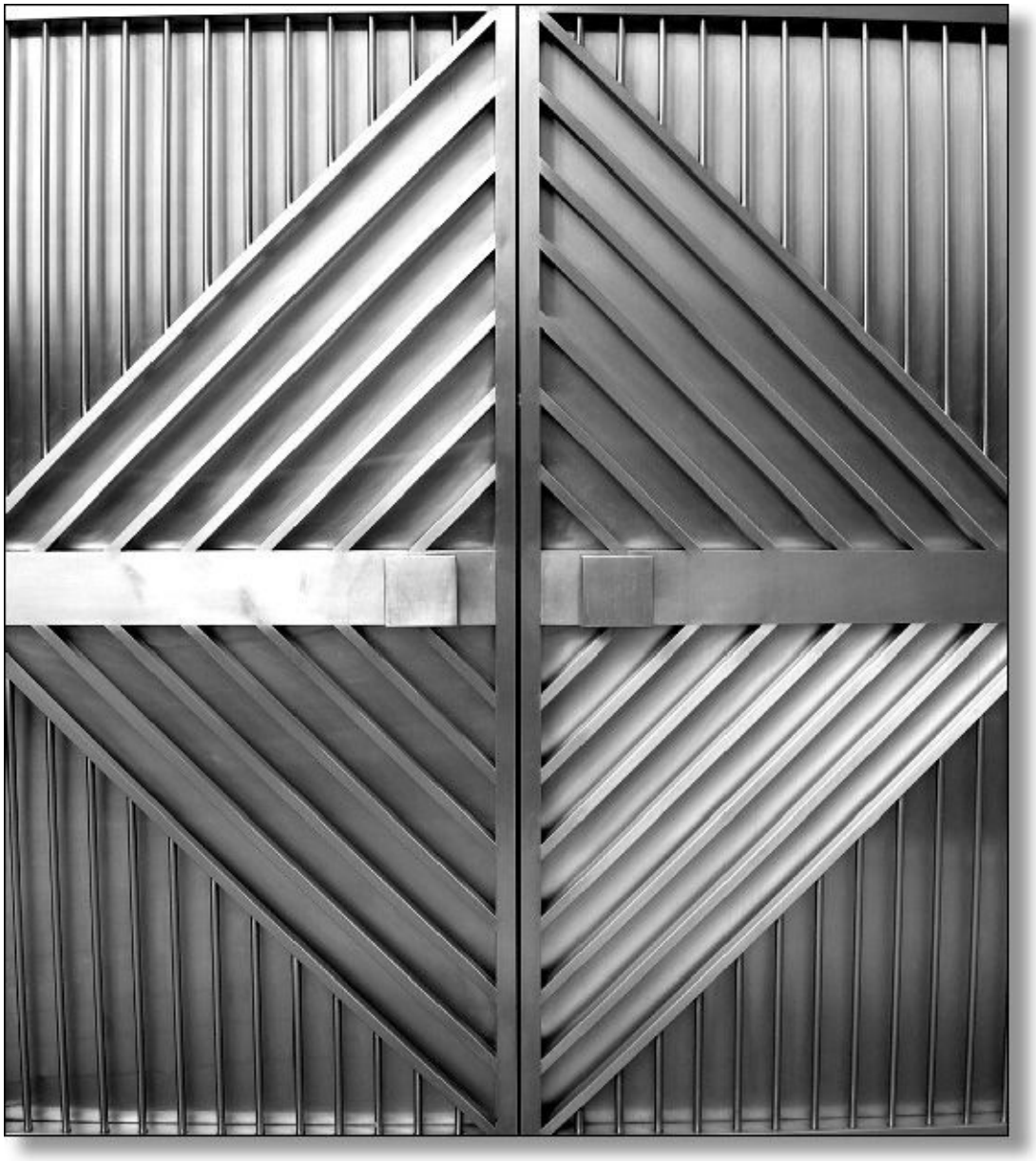
This way, the service consumer would be able to lock the resource for as long as it takes to read the state, modify it, and write it back again. Although other service consumers would still encounter exceptions while attempting to update the resource at the same time as the consumer that has locked it, it is deemed preferable to the

unpredictability of managing the resource as part of an optimistic locking model.

This solution is not embraced by all of the MUA architects because retaining the lock on the resource requires that the Stateless {308} constraint be breached. It could further lead to the danger of stale locks starting, impacting performance and scalability. In particular, unless proper measures are taken to ensure that only authorized consumers may lock a resource, this exposes the resources to denial of service attacks by malicious consumers that could lock out all other consumers.

After further discussion, a compromise is reached. The OptLock method will be attempted first. As a fallback, if the consumer tries three times and fails, it will attempt the stateful PesLock method to ensure it is able to complete the action.

Chapter 10. Service API and Contract Versioning with Web Services and REST Services



Note

This chapter provides a number of code examples that help demonstrate various versioning scenarios and approaches. Note that these code examples are not related to any code examples provided in Case Study Examples from preceding chapters.

After a service contract is deployed, consumer programs will naturally begin forming dependencies on it. When we are subsequently forced to make changes to the contract, we need to figure out:

- Whether the changes will negatively impact existing (and potentially future) service consumers
- How changes that will and will not impact consumers should be implemented and communicated

These issues result in the need for versioning. Anytime you introduce the concept of versioning into an SOA project, a number of questions will likely be raised, for example:

- What exactly constitutes a new version of a service contract? What's the difference between a major and minor version?
- What do the parts of a version number indicate?
- Will the new version of the contract still work with existing consumers that were designed for the old contract version?
- Will the current version of the contract work with new consumers that may have different data exchange requirements?
- What is the best way to add changes to existing contracts while minimizing the impact on consumers?
- Will we need to host old and new contracts at the same time? If yes, for how long?

We will address these questions and provide a set of options for solving common versioning problems. The upcoming sections begin by covering some basic concepts, terminology, and strategies specific to service contract versioning.

10.1 Versioning Basics

So when we say that we're creating a new version of a service contract, what

exactly are we referring to? The following sections explain some fundamental terms and concepts and further distinguish between Web service contracts and REST service contracts.

Versioning Web Services

As we've established many times in this book, a Web service contract can be comprised of several individual documents and definitions that are linked and assembled together to form a complete technical interface.

For example, a given Web service contract can consist of:

- One (sometimes more) WSDL definitions
- One (usually more) XML Schema definitions
- Some (sometimes no) WS-Policy definitions

Furthermore, each of these definition documents can be shared by other Web service contracts. For example,

- A centralized XML Schema definition will commonly be used by multiple WSDL definitions.
- A centralized WS-Policy definition will commonly be applied to multiple WSDL definitions.
- An abstract WSDL description can be imported by multiple concrete WSDL descriptions or vice versa.

Of all the different parts of a Web service contract, the part that establishes the fundamental technical interface is the abstract description of the WSDL definition. This represents the core of a Web service contract and is then further extended and detailed through schema definitions, policy definitions, and one or more concrete WSDL descriptions.

When we need to create a new version of a Web service contract, we can therefore assume that there has been a change in the abstract WSDL description or one of the contract documents that relates to the abstract WSDL description. The Web service contract content commonly subject to change is the XML schema content that provides the types for the abstract description's message definitions. Finally, the one other contract-related technology that can still impose versioning requirements but is less likely to do so simply because it is a less common part of Web service contracts is WS-Policy.

Versioning REST Services

If we follow the REST model of using a uniform contract to express service

capabilities, the sharing of definition documents between service contracts is even clearer. For example,

- All HTTP methods used in contracts are standard across the architecture.
- XML Schema definitions are standard, as they are wrapped up in general media types.
- The identifier syntax for lightweight service endpoints (known as resources) are standard across the architecture.

Changes to the uniform contract facets that underlie each service contract can impact any REST service in the service inventory.

Fine and Coarse-Grained Constraints

Regardless of whether XML schemas are used with Web services or REST services, versioning changes are often tied to the increase or reduction of the quantity or granularity of constraints expressed in the schema definition. Therefore, let's briefly recap the meaning of the term *constraint granularity* in relation to a type definition.

Note the bolded and italicized parts in Example 10.1:

Example 10.1 A `complexType` construct containing fine and coarse-grained constraints.

Click here to view code image

```
<xsd:element name="LineItem" type="LineItemType"/>
<xsd:complexType name="LineItemType">
  <xsd:sequence>
    <xsd:element name="productID" type="xsd:string"/>
    <xsd:element name="productName" type="xsd:string"/>
    <xsd:any minOccurs="0" maxOccurs="unbounded"
      namespace="##any" processContents="lax"/>
  </xsd:sequence>
  <xsd:anyAttribute namespace="##any"/>
</xsd:complexType>
```

As indicated by the bolded text, there are elements with specific names and data types that represent parts of the message definition with a *fine* level of constraint granularity. All the message instances (the actual XML documents that will be created based on this structure) must conform to these constraints to be considered valid (which is why these are considered the absolute "minimum" constraints).

The italicized text shows the element and attribute wildcards also contained by

this complex type. These represent parts of the message definition with an extremely *coarse* level of constraint granularity in that messages do not need to comply to these parts of the message definition at all.

The use of the terms “fine-grained” and “coarse-grained” is highly subjective. What may be a fine-grained constraint in one contract may not be in another. The point is to understand how these terms can be applied when comparing parts of a message definition or when comparing different message definitions with each other.

10.2 Versioning and Compatibility

The number one concern when developing and deploying a new version of a service contract is the impact it will have on other parts of the enterprise that have formed or will form dependencies on it. This measure of impact is directly related to how compatible the new contract version is with the old version and its surroundings in general.

This section establishes the fundamental types of compatibility that relate to the content and design of new contract versions and also tie into the goals and limitations of different versioning strategies introduced at the end of this chapter.

Backwards Compatibility

A new version of a service contract that continues to support consumer programs designed to work with the old version is considered *backwards compatible*. From a design perspective, this means that the new contract has not changed in such a way that it can impact existing consumer programs that are already using the contract.

Backwards Compatibility in Web Services

Example 10.2 provides a simple instance of a backwards-compatible change based on the addition of a new operation to an existing WSDL definition:

Example 10.2 The addition of a new operation represents a common backwards-compatible change.

Click here to view code image

```
<definitions name="Purchase Order" targetNamespace=
  "http://actioncon.com/contract/po"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://actioncon.com/contract/po"
  xmlns:po="http://actioncon.com/schema/po">
```

```
  ...
  <portType name="ptPurchaseOrder">
    <operation name="opSubmitOrder">
      <input message="tns:msgSubmitOrderRequest"/>
      <output message="tns:msgSubmitOrderResponse"/>
    </operation>
    <operation name="opCheckOrderStatus">
      <input message="tns:msgCheckOrderRequest"/>
      <output message="tns:msgCheckOrderResponse"/>
    </operation>
    <operation name="opChangeOrder">
      <input message="tns:msgChangeOrderRequest"/>
      <output message="tns:msgChangeOrderResponse"/>
    </operation>
    <operation name="opCancelOrder">
      <input message="tns:msgCancelOrderRequest"/>
      <output message="tns:msgCancelOrderResponse"/>
    </operation>
    <operation name="opGetOrder">
      <input message="tns:msgGetOrderRequest"/>
      <output message="tns:msgGetOrderResponse"/>
    </operation>
  </portType>
</definitions>
```

By adding a brand-new operation, we are creating a new version of the contract, but this change is backwards-compatible and will not impact any existing consumers. The new service implementation will continue to work with old service consumers because all the operations that an existing service consumer might invoke are still present and continue to meet the requirements of the previous service contract version.

Backwards Compatibility in REST Services

A backwards-compatible change to a REST-compliant service contract might involve adding some new resources or adding new capabilities to existing resources. In each of these cases the existing service consumers will only invoke the old methods on the old resources, which continue to work as they previously did.

As demonstrated in Example 10.3, supporting a new method that existing service consumers don’t use results in a backwards-compatible change. However, in a service inventory with multiple REST services, we can take steps to ensure that new service consumers will continue to work with old versions of services.

Example 10.3 The addition of a new resource or new supported method on a resource is a backwards-compatible change for a REST service.

Click here to view code image

```
Service: po.actioncon.com
Capabilities:
POST /orders
      In = application/vnd.com.actioncon.po+xml
GET orders{order-id}/status
      Out = text/plain
PUT orders{order-id}
      In = application/vnd.com/actioncon.po+xml
DELETE orders{order-id}
GET orders{order-id}
      Out = application/vnd.com.actioncon.po+xml
```

As shown in Example 10.4, it may be important for service consumers to have a reasonable way of proceeding with their interaction if the service reports that the new method is not implemented.

Example 10.4 New methods added to a service inventory's uniform contract need to provide a way for service consumers to "fall back" on a previously used method if they are to truly be backwards-compatible.

Click here to view code image

```
Legal methods for actioncon.com service inventory:
* GET
* PUT
* DELETE
* POST
* SUBSCRIBE (consumers must fall back to periodic GET if service
reports "not implemented")
```

Changes to schemas and media types approach backwards compatibility in a different manner, in that they describe how information can be encoded for transport, and will often be used in both request and response messages. The focus for backwards compatibility is on whether a new message recipient can make sense of information sent by a legacy source. In other words, the new processor must continue to understand information produced by a legacy message *generator*.

An example of a change made to a schema for a message definition that is backwards-compatible is the addition of an optional element (as shown in bolded markup code in Example 10.5).

Example 10.5 In an XML Schema definition, the addition of an optional element is also considered backwards-compatible.

Click here to view code image

```
Media type = application/vnd.com.actioncon.po+xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Here we are using a simplified version of the XML Schema definition for the Purchase Order service. The optional `available` element is added to the `LineItemType` complex type. This has no impact on existing generators because they are not required to provide this element in their messages. New processors must be designed to cope without the new information if they are to remain backwards-compatible.

Changing any of the existing elements in the previous example from required to optional (by adding the `minOccurs="0"` setting) would also be considered a backwards-compatible change. When we have control over how we choose to design the next version of a Web service contract, backwards compatibility is generally attainable. However, mandatory changes (such as those imposed by laws or regulations) can often force us to break backwards compatibility.

Note

Both the Flexible and Loose versioning strategies explained at the end of this chapter support backwards compatibility.

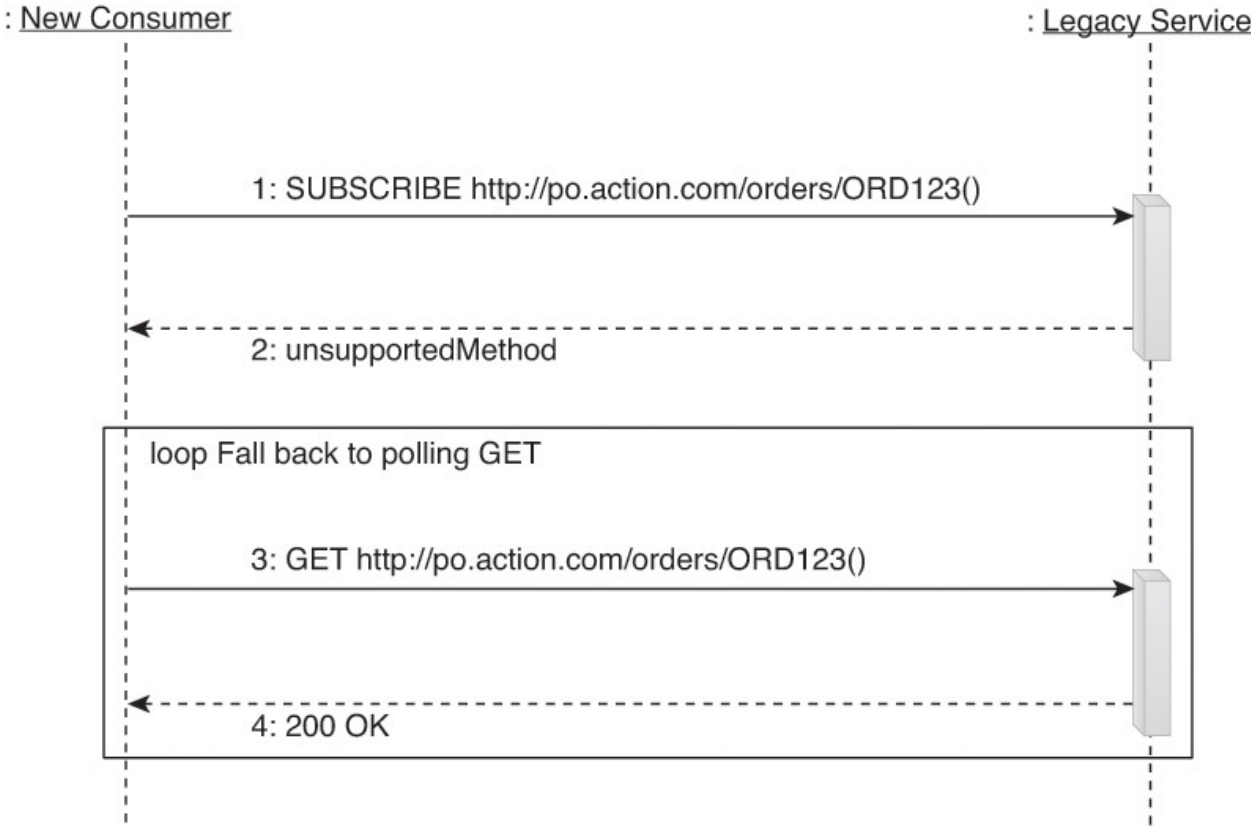
Forwards Compatibility

When a service contract is designed in such a manner so that it can support a range of future consumer programs, it is considered to have an extent of *forwards compatibility*. This means that the contract can essentially accommodate how consumer programs will evolve over time.

Supporting forwards compatibility for Web service operations or uniform contract methods requires exception types to be present in the contract to allow service consumers to recover if they attempt to invoke a new and unsupported

operation or method. For example, a “method not implemented” response enables the service consumer to detect that it is dealing with an incompatible service, thereby allowing it to handle this exception gracefully.

Redirection exception codes help REST services that implement a uniform contract change the resource identifiers in the contract when required. This is another way in which service contracts can allow legacy service consumers to continue using the service after contract changes have taken place (Example 10.6).



Example 10.6 A REST service ensures forwards compatibility by raising an exception whenever it does not understand a reusable contract or uniform contract method.

Forwards compatibility of schemas in REST services requires extension points to be present where new information can be added so that it will be safely ignored by legacy processors.

For example:

- Any validation that the processor does must not reject a document formatted according to the new schema.

- All existing information that the processor might need must remain present in future versions of the schema.
- Any new information added to the schema must be safe for legacy processors to ignore (if processors must understand the new information, then the change cannot be forwards compatible).
- The processor must ignore any information that it does not understand.

A common way to ensure validation does not reject future versions of the schema is to use wildcards in the earlier version. These provide extension points where new information can be added in future schema versions, as shown in Example 10.7.

Example 10.7 To support forwards compatibility within a message definition generally requires the use of XML Schema wildcards.

Click here to view code image

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:any namespace="##any" processContents="lax"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:anyAttribute namespace="##any"/>
  </xsd:complexType>
</xsd:schema>
```

In this example, the `xsd:any` and `xsd:anyAttribute` elements are added to allow for a range of unknown elements and data to be accepted by the service contract. In other words, the schema is being designed in advance to accommodate unforeseen changes in the future.

It is important to understand that building extension points into service contracts for forwards compatibility by no means eliminates the need to consider compatibility issues when making contract changes. New information can only be added to schemas in a forwards-compatible manner if it is genuinely safe for processors to ignore. New operations are only able to be made forwards-compatible if a service consumer has an existing operation to fall back on when it finds the one it initially attempted to invoke is unsupported.

A service with a forwards-compatible contract will often not be able to process all message content. Its contract is simply designed to accept a broader range of data unknown at the time of its design.

Note

Forwards compatibility forms the basis of the Loose versioning strategy that is explained shortly.

Compatible Changes

When we make a change to a service contract that does not negatively affect existing consumers, then the change itself is considered a *compatible change*.

Note

In this book, the term "compatible change" refers to backwards compatibility by default. When used in reference to forwards compatibility, it is further qualified as a *forwards-compatible change*.

A simple example of a compatible change is when we set the `minOccurs` attribute of an element from "1" to "0," effectively turning a required element into an optional one, as shown in Example 10.8.

Example 10.8 The default value of the `minOccurs` attribute is "1". Therefore because this attribute was previously absent from the `productName` element declaration, it was considered a required element. Adding the `minOccurs="0"` setting turns it into an optional element, resulting in a compatible change. (Note that making this change to a message output from the service would be an incompatible change.)

Click here to view code image

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"
        minOccurs="0"/>
```

```
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

This type of change will not impact existing consumer programs that are used to sending the element value to the Web service, nor will it affect future consumer programs that can be designed to optionally send that element.

Another example of a compatible change was provided earlier in Example 10.3, when we first added the optional `available` element declaration. Even though we extended the type with a whole new element, because it is optional it is considered a compatible change.

Here is a list of common compatible changes:

- Adding a new WSDL operation definition and associated message definitions
- Adding a new standard method to an existing REST resource
- Adding a set of new REST resources
- Changing the identifiers for a set of REST resources (including splitting and merging of services) using redirection response codes to facilitate migration of REST service consumers to the new identifiers
- Adding a new WSDL port type definition and associated operation definitions
- Adding new WSDL binding and service definitions
- Extending an existing uniform contract method in a way that can be safely ignored by REST services that can fall back on old service logic (for example, adding “If-None-Match” as a feature of the HTTP GET operation so that if the service ignores it, the consumer will still get the current and correct representation for the resource)
- Adding a new uniform contract method when an exception response exists for services that do not understand the method to use (and consumers can recover from this exception)
- Adding a new optional XML Schema element or attribute declaration to a message definition
- Reducing the constraint granularity of an XML Schema element or attribute of a message definition type used for input messages
- Adding a new XML Schema wildcard to a message definition type

- Adding a new optional WS-Policy assertion
- Adding a new WS-Policy alternative

Incompatible Changes

If after a change a contract is no longer compatible with consumers, then it is considered to have received an *incompatible change*. These are the types of changes that can break an existing contract and therefore impose the most challenges when it comes to versioning.

Note

The term “incompatible change” also indicates backwards compatibility by default. Incompatible changes that affect forwards compatibility will be qualified as “forwards-incompatible changes.”

Going back to our example, if we set an element’s `minOccurs` attribute from “0” to any number above zero, then we are introducing an incompatible change for input messages, as shown in Example 10.9:

Example 10.9 Incrementing the `minOccurs` attribute value of any established element declaration is automatically an incompatible change.

Click here to view code image

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"
        minOccurs="3"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="3"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

What was formerly an optional element is now required. This will certainly affect existing consumers that are not designed to comply with this new constraint, because adding a new required element introduces a mandatory constraint upon the contract.

Common incompatible changes include:

- Renaming an existing WSDL operation definition
- Removing an existing WSDL operation definition
- Changing the MEP of an existing WSDL operation definition
- Adding a fault message to an existing WSDL operation definition
- Adding a new required XML Schema element or attribute declaration to a message definition
- Increasing the constraint granularity of an XML Schema element or attribute declaration of a message definition
- Renaming an optional or required XML Schema element or attribute in a message definition
- Removing an optional or required XML Schema element or attribute or wildcard from a message definition
- Adding a new required WS-Policy assertion or expression
- Adding a new ignorable WS-Policy expression (most of the time)

Incompatible changes tend to cause most of the challenges with service contract versioning.

10.3 REST Service Compatibility Considerations

REST services within a given service inventory typically share a uniform contract for every resource, including uniform methods and media types. The same media types are used in both requests and responses, and new uniform contract facets are reused much more often than they are added to. This emphasis on service contract reuse within REST-compliant service inventories results in the need to highlight some special considerations, because changes to the uniform contract will automatically impact a range of service consumers because:

- The uniform contract methods are shared by all services.
- The uniform contract media types are shared by both services and service consumers.

As a result, both backwards compatibility and forwards compatibility considerations are almost equally important.

SOA Patterns

Service contracts that make use of the Schema Centralization [356]

pattern without necessarily being REST-compliant will often need to impose a similarly rigid view of forwards compatibility and backwards compatibility.

Uniform contract methods codify the kinds of interactions that can occur between services and their consumers. For example, GET codifies “fetch some data,” while PUT codifies “store some data.”

Because the kinds of interactions that occur between REST services within the same service inventory tend to be relatively limited and stable, methods will usually change at a low rate compared to media types or resources. Compatibility issues usually pertain to a set of allowable methods that are only changed after careful case-by-case consideration.

An example of a compatible change to HTTP is the addition of `If-None-Match` headers to GET requests. If a service consumer knows the last version (or `etag`) of the resource that it fetched, it can make its GET request conditional. The `If-None-Match` header allows the consumer to state that the GET request should not be executed if the version of the resource is still the same as it was for the consumer’s last fetch. Instead, it will return the normal GET response, although it will do so in a non-optimal mode.

An example of an incompatible change to HTTP is the addition of a `Host` header used to support multihoming of Web servers. HTTP/1.0 did not require the name of the service to be included in request messages, but HTTP/1.1 does require this. If the special `Host` header is missing, HTTP/1.1 services must reject the request as being badly formed. However, HTTP/1.1 services are also required to be backwards-compatible, so if an HTTP/1.0 request comes into the REST service it will still be handled according to HTTP/1.0 rules.

Uniform contract media types further codify the kinds of information that can be exchanged between REST services and consumers. As previously stated, media types tend to change at a faster rate than HTTP methods in the uniform contract; however, media types still change more slowly than resources. Compatible change is more of a live concern for the media types, and we can draw some more general rules about how to deal with them.

For example, if the generator of a message indicates to a processor of the message that it conforms to a particular media type, the processor generally does not need to know which version of the schema was used, nor does the processor need to have been built against the same version of the schema. The processor expects that all versions of the schema for a particular media type will be both

forwards compatible and backwards-compatible with the type it was developed to support. Likewise, the generator expects that when it produces a message conformant with a particular schema version, that all processors of the message will understand it.

When incompatible changes are made to a schema, a new media type identifier is generally required to ensure that:

- The processor can decide how to parse a document based on the media type identifier
- Services and consumers are able to *negotiate* for a specific media type that will be understood by the processor when the message has been produced

Content negotiation is the ultimate fallback to ensure compatibility in REST-compliant service inventories. For a fetch interaction this often involves the consumer indicating to the service what media types it is able to support, and the service returning the most appropriate type that it supports. This mechanism allows for incompatible changes to be made to media types, as required.

Note

One way to better understand versioning issues that pertain to media types is to look at how they are used in HTML. An example of a compatible change to HTML that did not result in the need for a new media type was the addition of the `abbr` element to version 4.0 of the HTML language. This element allows new processors of HTML documents to support a mouse-over to expand abbreviations on a web page and to better support accessibility of the page. Legacy processors safely ignore the expansion, but will continue correctly showing the abbreviation itself.

An example of an incompatible change to HTML that did require a new media type was the conversion of HTML 4.0 to XML (resulting in version 1.0 of XHTML). The media type for the traditional SGML version remained `text/html`, while the XML version became `application/xhtml+xml`. This allowed content negotiation to occur between the two types, and for processors to choose the correct parser and validation strategy based on which type was specified by the service.

Some incompatible changes have also been made to HTML without changing the media type. HTML 4.0 deprecated APPLET, BASEFONT, CENTER, DIR, FONT, ISINDEX, MENU, S,

STRIKE, and U elements in favor of newer elements. These elements must continue to be understood but their use in HTML documents is being phased out. HTML 4.0 made LISTING, PLAINTEXT, and XMP obsolete. These elements should not be used in HTML 4.0 documents and no longer need to be understood.

Deprecating elements over a long period of time and eventually identifying them as obsolete once they are no longer used by existing services or consumers is a technique that can be used for REST media types to incrementally update a schema without having to change the media type.

10.4 Version Identifiers

One of the most fundamental design patterns related to Web service contract design is the Version Identification pattern. It essentially advocates that version numbers should be clearly expressed, not just at the contract level, but right down to the versions of the schemas that underlie the message definitions.

The first step to establishing an effective versioning strategy is to decide on a common means by which versions themselves are identified and represented within Web service contracts.

Versions are almost always communicated with version numbers. The most common format is a decimal, followed by a period and then another decimal, as shown here:

```
version="2.0"
```

Sometimes, you will see additional period plus decimal pairs that lead to more detailed version numbers like this:

```
version="2.0.1.1"
```

The typical meaning associated with these numbers is the measure or significance of the change. Incrementing the first decimal generally indicates a major version change (or upgrade) in the software, whereas decimals after the first period usually represent various levels of minor version changes.

From a compatibility perspective, we can associate additional meaning to these numbers. Specifically, the following convention has emerged in the industry:

- A minor version is expected to be backwards-compatible with other minor versions associated with a major version. For example, version 5.2 of a program should be fully backwards-compatible with versions 5.0 and 5.1.

- A major version is generally expected to break backwards compatibility with programs that belong to other major versions. This means that program version 5.0 is not expected to be backwards-compatible with version 4.0.

Note

A third “patch” version number is also sometimes used to express changes that are both forwards-compatible and backwards-compatible. Typically these versions are intended to clarify the schema only, or to fix problems with the schema that were discovered after it was deployed. For example, version 5.2.1 is expected to be fully compatible with version 5.2.0, but may be added for clarification purposes.

This convention of indicating compatibility through major and minor version numbers is referred to as the *compatibility guarantee*. Another approach, known as “amount of work,” uses version numbers to communicate the effort that has gone into the change. A minor version increase indicates a modest effort, and a major version increase predictably represents a lot of work.

These two conventions can be combined and often are. The result is often that version numbers continue to communicate compatibility as explained earlier, but they sometimes increment by several digits, depending on the amount of effort that went into each version.

There are various syntax options available to express version numbers. For example, you may have noticed that the declaration statement that begins an XML document can contain a number that expresses the version of the XML specification being used:

```
<?xml version="1.0"?>
```

That same `version` attribute can be used with the root `xsd:schema` element, as follows:

Click here to view code image

```
<xsd:schema version="2.0" ...>
```

You can further create a custom variation of this attribute by assigning it to any element you define (in which case you are not required to name the attribute “version”).

```
<LineItem version="2.0">
```

An alternative custom approach is to embed the major version number into a namespace or media type identifier, as shown here:

Click here to view code image

```
<LineItem xmlns="http://actioncon.com/schema/po/v2">
```

or

Click here to view code image

```
application/vnd.com.actioncon.po.v2+xml
```

Note that it has become a common convention to use date values in namespaces when versioning XML schemas, as follows:

Click here to view code image

```
<LineItem xmlns="http://actioncon.com/schema/po/2010/09">
```

In this case, it is the date of the change that acts as the major version identifier. To keep the expression of XML Schema definition versions in alignment with WSDL definition versions, we use version numbers instead of date values in upcoming examples. However, when working in an environment where XML Schema definitions are separately owned as part of an independent data architecture, it is not uncommon for schema versioning identifiers to be different from those used by WSDL definitions.

Regardless of which option you choose, it is important to consider the Canonical Versioning pattern that dictates that the expression of version information must be standardized across all service contracts within the boundary of a service inventory. In larger environments, this will often require a central authority that can guarantee the linearity, consistency, and description quality of version information. These types of conventions carry over into how service termination information is expressed, as further explored in Chapter 23 in *Web Service Contract Design and Versioning for SOA*.

SOA Patterns

Of course you may also be required to work with third-party schemas and WSDL definitions that may already have implemented their own versioning conventions. In this case, the extent to which the Canonical Versioning [327] pattern can be applied will be limited.

10.5 Versioning Strategies

There is no one versioning approach that is right for everyone. Because versioning represents a governance-related phase in the overall lifecycle of a service, it is a practice that is subject to the conventions, preferences, and requirements that are distinct to any enterprise.

Even though there is no de facto versioning technique for the WSDL, XML Schema, and WS-Policy content that comprises Web service contracts, a number of common and advocated versioning approaches have emerged, each with its own benefits and tradeoffs.

In this section, we single out the following three common strategies:

- *Strict* – Any compatible or incompatible changes result in a new version of the service contract. This approach does not support backwards or forwards compatibility.
- *Flexible* – Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility but not forwards compatibility.
- *Loose* – Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility and forwards compatibility.

These strategies are explained individually in the upcoming sections.

The Strict Strategy (New Change, New Contract)

The simplest approach to Web service contract versioning is to require that a new version of a contract be issued whenever any kind of change is made to any part of the contract.

This is commonly implemented by changing the target namespace value of a WSDL definition (and possibly the XML Schema definition) every time a compatible or incompatible change is made to the WSDL, XML Schema, or WS-Policy content related to the contract. Namespaces are used for version identification instead of a version attribute because changing the namespace value automatically forces a change in all consumer programs that need to access the new version of the schema that defines the message types.

This “super-strict” approach is not really that practical, but it is the safest and sometimes warranted when there are legal implications to Web service contract modifications, such as when contracts are published for certain interorganization data exchanges. Because both compatible and incompatible changes will result

in a new contract version, this approach supports neither backwards nor forwards compatibility.

Pros and Cons

The benefit of this strategy is that you have full control over the evolution of the service contract, and because backwards and forwards compatibility are intentionally disregarded, you do not need to concern yourself with the impact of any change in particular (because all changes effectively break the contract).

On the downside, by forcing a new namespace upon the contract with each change, you are guaranteeing that all existing service consumers will no longer be compatible with any new version of the contract. Consumers will only be able to continue communicating with the Web service while the old contract remains available alongside the new version or until the consumers themselves are updated to conform to the new contract.

Therefore, this approach will increase the governance burden of individual services and will require careful transitioning strategies. Having two or more versions of the same service co-exist at the same time can become a common requirement for which the supporting service inventory infrastructure needs to be prepared.

The Flexible Strategy (Backwards Compatibility)

A common approach used to balance practical considerations with an attempt at minimizing the impact of changes to Web service contracts is to allow compatible changes to occur without forcing a new contract version, while not attempting to support forwards compatibility at all.

This means that any backwards-compatible change is considered safe in that it ends up extending or augmenting an established contract without affecting any of the service's existing consumers. A common example of this is adding a new operation to a WSDL definition or adding an optional element declaration to a message's schema definition.

As with the Strict strategy, any change that breaks the existing contract does result in a new contract version, usually implemented by changing the target namespace value of the WSDL definition and potentially also the XML Schema definition.

Pros and Cons

The primary advantage to this approach is that it can be used to accommodate a

variety of changes while consistently retaining the contract's backwards compatibility. However, when compatible changes are made, these changes become permanent and cannot be reversed without introducing an incompatible change. Therefore, a governance process is required during which each proposed change is evaluated so that contracts do not become overly bloated or convoluted. This is an especially important consideration for agnostic services that are heavily reused.

The Loose Strategy (Backwards and Forwards Compatibility)

As with the previous two approaches, this strategy requires that incompatible changes result in a new service contract version. The difference here is in how service contracts are initially designed.

Instead of accommodating known data exchange requirements, special features from the WSDL, XML Schema, and WS-Policy languages are used to make parts of the contract intrinsically extensible so that they remain able to support a broad range of future, unknown data exchange requirements. For example:

- The `anyType` attribute value provided by the WSDL 2.0 language allows a message to consist of any valid XML document.
- XML Schema wildcards can be used to allow a range of unknown data to be passed in message definitions.
- Ignorable policy assertions can be defined to communicate service characteristics that can optionally be acknowledged by future consumers.

These and other features related to forwards compatibility are discussed in *Web Service Contract Design and Versioning for SOA*.

Pros and Cons

The fact that wildcards allow undefined content to be passed through Web service contracts provides a constant opportunity to further expand the range of acceptable message element and data content. On the other hand, the use of wildcards will naturally result in vague and overly coarse service contracts that place the burden of validation on the underlying service logic.

Strategy Summary

Provided in Table 10.1 is a broad summary of how the three strategies compare based on three fundamental characteristics.

	Strategy		
	Strict	*Flexible*	*Loose*
Strictness	High	Medium	Low
Governance Impact	High	Medium	High
Complexity	Low	Medium	High

Table 10.1 A general comparison of the three versioning strategies.

The three characteristics used in this table to form the basis of this comparison are as follows:

- *Strictness* – The rigidity of the contract versioning options. The Strict approach clearly is the most rigid in its versioning rules, while the Loose strategy provides the broadest range of versioning options due to its reliance on wildcards.
- *Governance Impact* – The amount of governance burden imposed by a strategy. Both Strict and Loose approaches increase governance impact but for different reasons. The Strict strategy requires the issuance of more new contract versions, which impacts surrounding consumers and infrastructure, while the Loose approach introduces the concept of unknown message sets that need to be separately accommodated through custom programming.
- *Complexity* – The overall complexity of the versioning process. Due to the use of wildcards and unknown message data, the Loose strategy has the highest complexity potential, while the straightforward rules that form the basis of the Strict approach make it the simplest option.

Throughout this comparison, the Flexible strategy provides an approach that represents a consistently average level of strictness, governance effort, and overall complexity.

10.6 REST Service Versioning Considerations

REST services that share the same uniform contract maintain separate versioned specifications for the following:

- The version number or specification of the resource identifier syntax (as per the “Request for Comments 6986 - Uniform Resource Identifier (URI):

Generic Syntax” specification)

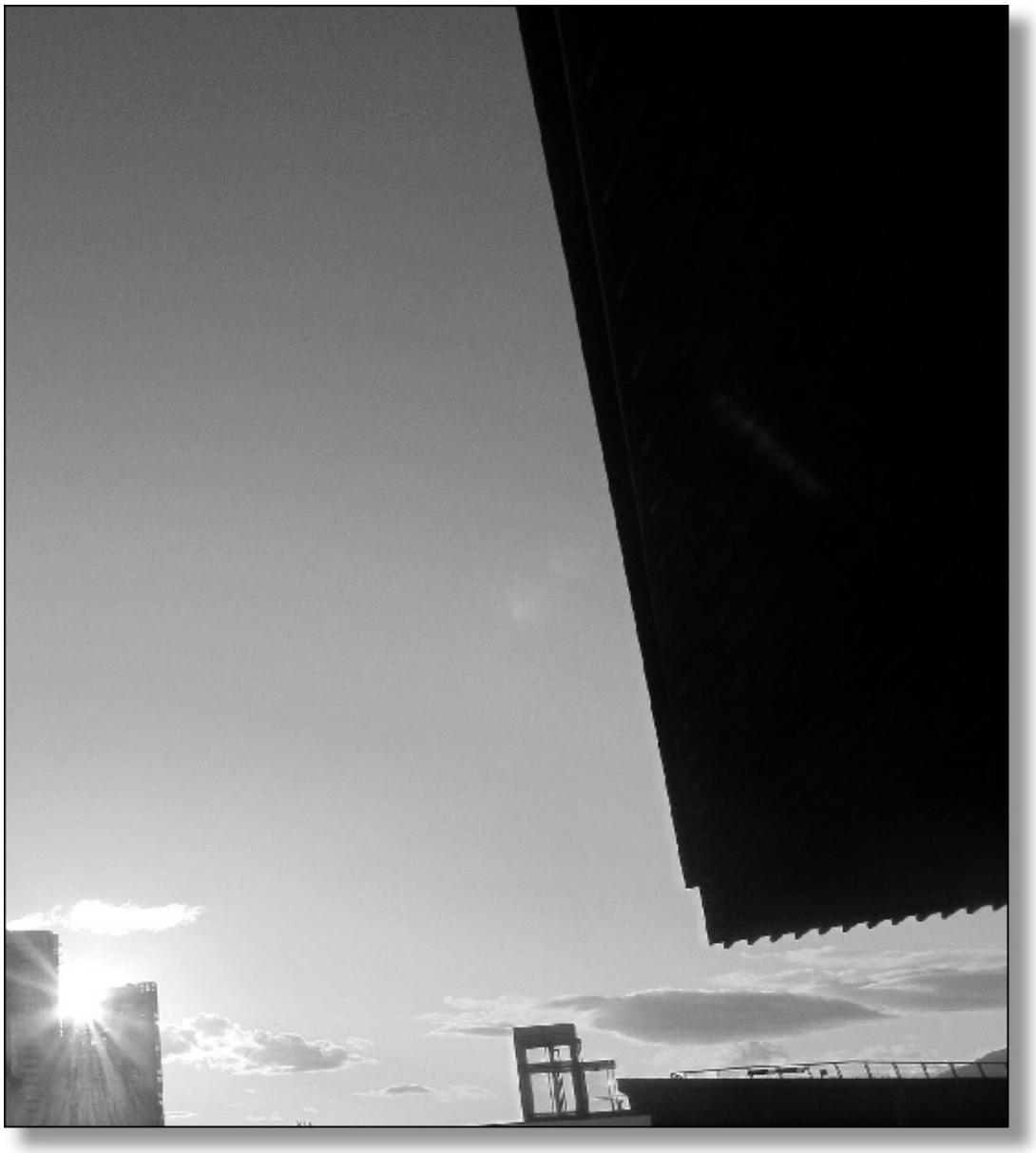
- The specification of the collection of legal methods, status codes, and other interaction protocol details (as per the “Request for Comments 2616 - Hypertext Transfer Protocol - HTTP/1.1” specification)
- Individual specifications for legal media types (for example. HTML 4.01 and the “Request for Comments 4287 - The Atom Syndication Format” specification)
- Individual specifications for service contracts that use the legal resource identifier syntax, methods, and media types

Each part of the uniform contract is specified and versioned independently of the others. Changing any one specification does not generally require another specification to be updated or versioned. Likewise, changing any of the uniform contract facet specifications does not require changes to individual service contracts, or changes to their version numbers.

This last point is in contradiction to some conventional versioning strategies. One might expect that if a schema used in a service contract changed, then the service contract would need to be modified. However, with REST services there is a tendency to maintain both forwards compatibility and backwards compatibility. If a REST service consumer sends a message that conforms to a newer schema, the service can process it as if it conformed to the older schema. If compatibility between these schemas has been maintained, then the service will function correctly. Likewise, if the service returns a message to the consumer that conforms to an old schema, the newer service consumer can still process the message correctly.

REST service contracts only need to directly consider the versioning of the uniform contract when media types used become deprecated, or when the schema advances so far that elements and attributes the service depends on are on their way to becoming obsolete. When this occurs, the service contract needs to be updated, and with it, the underlying service logic that processes the media types.

Part III: Appendices



Appendix A. ServiceOrientation Principles Reference

This appendix provides profile tables for the serviceorientation principles referenced throughout this book. As explained in Chapter 1, each principle reference is suffixed with the page number of its corresponding profile table in this appendix.

Every profile table contains the following sections:

- *Short Definition* – A concise, single-statement definition that establishes the fundamental purpose of the principle.

- *Long Definition* – A longer description of the principle that provides more detail as to what it is intended to accomplish.
- *Goals* – A list of specific design goals that are expected from the application of the principle. Essentially, this list provides the ultimate results of the principle's realization.
- *Design Characteristics* – A list of specific design characteristics that can be realized via the application of the principle. This provides some insight as to how the principle ends up shaping the service.
- *Implementation Requirements* – A list of common prerequisites for effectively applying the design principle. These can range from technology to organizational requirements.

Note that these profile tables provide only summarized versions of the principles. Complete coverage of the eight serviceorientation design principles, including case studies, is provided in the *SOA Principles of Service Design* book.

For more information about this and other titles in the *Prentice Hall Service Technology Series from Thomas Erl*, visit www.servicetechbooks.com. Summarized content of topics related to serviceorientation can also be found online at www.serviceorientation.com.

Standardized Service Contract	
Short Definition	*"Services share standardized contracts."*
Long Definition	*"Services within the same service inventory are in compliance with the same contract design standards."*
Goals	• To enable services with a meaningful level of natural interoperability within the boundary of a service inventory. This reduces the need for data transformation because consistent data models are used for information exchange. • To allow the purpose and capabilities of services to be more easily and intuitively understood. The consistency with which service functionality is expressed through service contracts increases interpretability and the overall predictability of service endpoints throughout a service inventory. Note that these goals are further supported by other service-orientation principles as well.
Design Characteristics	• A service contract (comprised of a technical interface or one or more service description documents) is provided with the service. • The service contract is standardized through the application of design standards.

Implementation Requirements	The fact that contracts need to be standardized can introduce significant implementation requirements to organizations that do not have a history of using standards. For example: • Design standards and conventions need to ideally be in place prior to the delivery of any service in order to ensure adequately scoped standardization. (For those organizations that have already produced ad-hoc Web services, retro-fitting strategies may need to be employed.) • Formal processes need to be introduced to ensure that services are modeled and designed consistently, incorporating accepted design principles, conventions, and standards.
	• Because achieving standardized service contracts generally requires a "contract-first" approach to service-oriented design, the full application of this principle will often demand the use of development tools capable of importing a customized service contract without imposing changes. • Appropriate skill sets are required to carry out the modeling and design processes with the chosen tools. When working with Web services, the need for a high level of proficiency with XML schema and WSDL languages is practically unavoidable. WS-Policy expertise may also be required. These and other requirements can add up to a noticeable transition effort that goes well beyond technology adoption.

Table A.1 A profile for the Standardized Service Contract principle

Service Loose Coupling	
Short Definition	*"Services are loosely coupled."*
Long Definition	*"Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment."*
Goals	By consistently fostering reduced coupling within and between services, we are working toward a state where service contracts increase independence from their implementations and services are increasingly independent from each other. This promotes an environment in which services and their consumers can be adaptively evolved over time with minimal impact on each other.
Design Characteristics	• The existence of a service contract that is ideally decoupled from technology and implementation details. • A functional service context that is not dependent on outside logic. • Minimal consumer coupling requirements.
Implementation Requirements	• Loosely coupled services are typically required to perform more runtime processing than if they were more tightly coupled. As a result, data exchange in general can consume more runtime resources, especially during concurrent access and high usage scenarios. • Achieving the right balance of coupling, while also supporting the other service-orientation principles that affect contract design, requires increased service contract design proficiency.

Table A.2 A profile for the Service Loose Coupling principle

Service Abstraction	
Short Definition	*"Non-essential service information is abstracted."*
Long Definition	*"Service contracts only contain essential information and information about services is limited to what is published in service contracts."*
Goals	Many of the other principles emphasize the need to publish *more* information in the service contract. The primary role of this principle is to keep the quantity and detail of contract content concise and balanced and prevent unnecessary access to additional service details.
Design Characteristics	• Services consistently abstract specific information about technology, logic, and function away from the outside world (the world outside of the service boundary). • Services have contracts that concisely define interaction requirements and constraints and other required service meta details. • Outside of what is documented in the service contract, information about a service is controlled or altogether hidden within a particular environment.
Implementation Requirements	The primary prerequisite to achieving the appropriate level of abstraction for each service is the level of service contract design skill applied.

Table A.3 A profile for the Service Abstraction principle

Service Reusability	
Short Definition	*"Services are reusable."*
Long Definition	*"Services contain and express agnostic logic and can be positioned as reusable enterprise resources."*
Goals	The goals behind Service Reusability are tied directly to some of the most strategic objectives of service-oriented computing: • To allow for service logic to be repeatedly leveraged over time so as to achieve an increasingly high return on the initial investment of delivering the service. • To increase business agility on an organizational level by enabling the rapid fulfillment of future business automation requirements through wide-scale service composition. • To enable the realization of agnostic service models. • To enable the creation of service inventories with a high percentage of agnostic services.
Design Characteristics	• The logic encapsulated by the service is associated with a context that is sufficiently agnostic to any one usage scenario so as to be considered reusable. • The logic encapsulated by the service is sufficiently generic, allowing it to facilitate numerous usage scenarios by different types of service consumers. • The service contract is flexible enough to process a range of input and output messages. • Services are designed to facilitate simultaneous access by multiple consumer programs.

Implementation Requirements	From an implementation perspective, Service Reusability can be the most demanding of the principles we've covered so far. Below are common requirements for creating reusable services and supporting their long-term existence: • A scalable runtime hosting environment capable of high-to-extreme concurrent service usage. Once a service inventory is relatively mature, reusable services will find themselves in an increasingly large number of compositions. • A solid version control system to properly evolve contracts representing reusable services. • Service analysts and designers with a high degree of subject matter expertise who can ensure that the service boundary and contract accurately represent the service's reusable functional context. • A high level of service development and commercial software development expertise so as to structure the underlying logic into generic and potentially decomposable components and routines. These and other requirements place an emphasis on the appropriate staffing of the service delivery team, as well as the importance of a powerful and scalable hosting environment and supporting infrastructure.

Table A.4 A profile for the Service Reusability principle

Service Autonomy	
Short Definition	*"Services are autonomous."*
Long Definition	*"Services exercise a high level of control over their underlying runtime execution environment."*
Goals	• To increase a service's runtime reliability, performance, and predictability, especially when being reused and composed. • To increase the amount of control a service has over its runtime environment. By pursuing autonomous design and runtime environments, we are essentially aiming to increase post-implementation control over the service and the service's control over its own execution environment.
Design Characteristics	• Services have a contract that expresses a well-defined functional boundary that should not overlap with other services. • Services are deployed in an environment over which they exercise a great deal (and preferably an exclusive level) of control. • Service instances are hosted by an environment that accommodates high concurrency for scalability purposes.
Implementation Requirements	• A high level of control over how service logic is designed and developed. Depending on the level of autonomy being sought, this may also involve control over the supporting data models. • A distributable deployment environment, so as to allow the service to be moved, isolated, or composed as required. • An infrastructure capable of supporting desired autonomy levels.

Table A.5 A profile for the Service Autonomy principle

Service Statelessness	
Short Definition	*"Services minimize statefulness."*
Long Definition	*"Services minimize resource consumption by deferring the management of state information when necessary."*
Goals	• To increase service scalability. • To support the design of agnostic service logic and improve the potential for service reuse.
Design Characteristics	What makes this somewhat of a unique principle is the fact that it is promoting a condition of the service that is temporary in nature. Depending on the service model and state deferral approach used, different types of design characteristics can be implemented. Some examples include: • Highly business process-agnostic logic so that the service is not designed to retain state information for any specific parent business process. • Less constrained service contracts so as to allow for the receipt and transmission of a wider range of state data at runtime. • Increased amounts of interpretive programming routines capable of parsing a range of state information delivered by messages and responding to a range of corresponding action requests.
Implementation Requirements	Although state deferral can reduce the overall consumption of memory and system resources, services designed with statelessness considerations can also introduce some performance demands associated with the runtime retrieval and interpretation of deferred state data.

	Here is a short checklist of common requirements that can be used to assess the support of stateless service designs by vendor technologies and target deployment locations: • The runtime environment should allow for a service to transition from an idle state to an active processing state in a highly efficient manner.
	• Enterprise-level or high-performance XML parsers and hardware accelerators (and SOAP processors) should be provided to allow services implemented as Web services to more efficiently parse larger message payloads with less performance constraints. • The use of attachments may need to be supported by Web services to allow messages to include bodies of payload data that do not undergo interface-level validation or translation to local formats. The nature of the implementation support required by the average stateless service in an environment will depend on the state deferral approach used within the service-oriented architecture.

Table A.6 A profile for the Service Statelessness principle

Service Discoverability	
Short Definition	*"Services are discoverable."*
Long Definition	*"Services are supplemented with communicative metadata by which they can be effectively discovered and interpreted."*
Goals	• Services are positioned as highly discoverable resources within the enterprise. • The purpose and capabilities of each service are clearly expressed so that they can be interpreted by humans and software programs. Achieving these goals requires foresight and a solid understanding of the nature of the service itself. Depending on the type of service model being designed, realizing this principle may require both business and technical expertise.
Design Characteristics	• Service contracts are equipped with appropriate metadata that will be correctly referenced when discovery queries are issued. • Service contracts are further outfitted with additional meta information that clearly communicates their purpose and capabilities to humans.
	• If a service registry exists, registry records are populated with the same attention to meta information as just described. • If a service registry does not exist, service profile documents are authored to supplement the service contract and to form the basis for future registry records.
Implementation Requirements	• The existence of design standards that govern the meta information used to make service contracts discoverable and interpretable, as well as guidelines for how and when service contracts should be further supplemented with annotations. • The existence of design standards that establish a consistent means of recording service meta information outside of the contract. This information is either collected in a supplemental document in preparation for a service registry, or is placed in the registry itself. You may have noticed the absence of a service registry on the list of implementation requirements. As previously established, the goal of this principle is to implement design characteristics within the service, not within the architecture.

Table A.7 A profile for the Service Discoverability principle

Service Composability	
Short Definition	*"Services are composable."*
Long Definition	*"Services are effective composition participants, regardless of the size and complexity of the composition."*
Goals	When discussing the goals of Service Composability, most of the goals of Service Reusability apply. This is because service composition often turns out to be a form of service reuse. In fact, you may recall that one of the objectives we listed for the Service Reusability principle was to enable wide-scale service composition. However, above and beyond simply attaining reuse, service composition provides the medium through which we can achieve what is often classified as the ultimate goal of service-oriented computing. By establishing an enterprise comprised of solution logic represented by an inventory of highly reusable services, we provide the means for a large extent of future business automation requirements to be fulfilled through service composition.
Design Characteristics For Composition Member Capabilities	Ideally, every service capability (especially those providing reusable logic) is considered a potential composition member. This essentially means that the design characteristics already established by the Service Reusability principle are equally relevant to building effective composition members. Additionally, there are two further characteristics emphasized by this principle: • The service needs to possess a highly efficient execution environment. More so than being able to manage concurrency, the efficiency with which composition members perform their individual processing should be highly tuned.

	• The service contract needs to be flexible so that it can facilitate different types of data exchange requirements for similar functions. This typically relates to the ability of the contract to exchange the same type of data at different levels of granularity. The manner in which these qualities go beyond mere reuse has to do primarily with the service being capable of optimizing its runtime processing responsibilities in support of multiple, simultaneous compositions.
Design Characteristics for Composition Controller Capabilities	Composition members will often also need to act as controllers or sub-controllers within different composition configurations. However, services designed as designated controllers are generally alleviated from many of the high-performance demands placed on composition members. These types of services therefore have their own set of design characteristics: • The logic encapsulated by a designated controller will almost always be limited to a single business task. Typically, the task service model is used, resulting in the common characteristics of that model being applied to this type of service.
	• While designated controllers may be reusable, service reuse is not usually a primary design consideration. Therefore, the design characteristics fostered by Service Reusability are considered and applied where appropriate, but with less of the usual rigor applied to agnostic services. • Statelessness is not always as strictly emphasized on designated controllers as with composition members. Depending on the state deferral options available by the surrounding architecture, designated controllers may sometimes need to be designed to remain fully stateful while the underlying composition members carry out their respective parts of the overall task. Of course, any capability acting as a controller can become a member of a larger composition, which brings the previously listed composition member design characteristics into account as well.

Table A.8 A profile for the Service Composability principle

Appendix B. REST Constraints Reference

This appendix provides profile tables for the REST constraints referenced throughout this book. As explained in Chapter 1, each constraint reference is suffixed with the page number of its corresponding profile table in this appendix.

Every profile table contains the following sections:

- *Short Definition* – A concise, single-statement definition that establishes the fundamental purpose of the constraint.
- *Long Definition* – A longer description of the constraint that provides more

detail as to what it is intended to accomplish.

- *Application* – A list of common steps and requirements for applying the constraint.
- *Impacts* – A list of positive and negative impacts that can result from the application of the constraint.
- *Relationship to REST* – A brief explanation of how the constraint can relate to other constraints and overall REST architecture.
- *Related REST Goals* – A list of REST design goals that are related to and relevant to the application of this constraint.
- *Related Service-Orientation Principles* – A list of service-orientation principles related to the constraint.
- *Related SOA Patterns* – A list of SOA design patterns related to the constraint.

Note that these profile tables provide only summarized versions of the constraints. Complete coverage of the REST constraints, including case studies, is provided in the *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST* book.

For more information about this and other titles in the *Prentice Hall Service Technology Series from Thomas Erl*, visit www.servicetechbooks.com. Summarized content of REST-related topics can also be found online at www.whatisrest.com.

Client-Server	
Short Definition	*"Solution logic is separated into consumer and service logic that share a technical contract."*
Long Definition	*"Business automation logic is organized into a solution comprised of units of consumer and service logic. Service consumers actively invoke service capabilities by sending messages that comply with a published technical service contract. Services passively wait to process request messages and respond to their receipt in compliance with the technical contract."*
Application	• Solution logic must undergo a process whereby it is subjected to the separation of concerns. This partitions the logic into units that address defined concerns. These units of logic are composed to form the solution at runtime. • The consumer's required knowledge about a service and the service's required knowledge of its consumers are limited to the contents of the shared technical contract.
Impacts	• Service logic can become more scalable and reusable because it is freed from having to implement consumer-specific logic. • Service and consumer logic are simplified due to respective information hiding. • Service and consumer implementations can be evolved independently in ways that do not require alterations to the shared contract.

	• Interactions between services and consumers that circumvent the shared technical contract are prohibited, potentially resulting in lost opportunities to optimize the solution architecture.
Relationship to REST	This is a foundational constraint that defines the separation between service, consumer, and the technical contract they share. All of the other constraints reference these artifacts and so build upon this constraint.
Related REST Goals	Modifiability, Scalability
Related Service-Orientation Principles	Service Loose Coupling (293), Service Abstraction (294)
Related SOA Patterns	Capability Composition [328], Contract Denormalization [335], Decoupled Contract [337], Functional Decomposition [344]

Stateless	
Short Definition	*"Services remain stateless between request/response message exchanges with service consumers."*
Long Definition	*"The communication between a service and a consumer is regulated so that the consumer provides all data necessary for the service to understand each consumer request. Between requests, the service is not permitted to retain any state data specific to its interaction with the consumer instance, allowing it to exist in a stateless condition. Instead, session state is deferred to the consumer at the end of each request."*
Application	• Consumer logic must be designed to preserve state data between requests and to issue request messages containing state data. • The request message must contain all of the state data necessary for the service to process the request, and the service must be able to "forget" the state data upon issuing the response without compromising the overall interaction. • Because the service is only involved in the automation of a solution when a consumer is actively making a request to it, in-between requests the service is "at rest," and therefore using no CPU, memory, or network resources on behalf of the consumer. • The service cannot be required to store data specific to a run-time instance of a service consumer. However, the service is still allowed to store data that is related to its own functional context.
Impacts	• Making consumers responsible for preserving state data alleviates the service from having to store and replicate potentially volatile data that is only relevant to the individual consumer program.

	• Deferring session state to consumers between requests frees up service memory resources, allowing the service to scale with the number of concurrent requests, rather than with the total number of concurrent consumers. • Messages can be understood by the service without the need to have inspected earlier messages. This can simplify service logic design and further reduce the complexity of debugging. • The requirement to repeatedly transmit potentially redundant state data can increase network traffic and processing overhead. • Reliability of state data can be both positively and negatively impacted: Service instance failures can be dealt with gracefully because the service does not retain state, but failure of the service consumer can result in a loss of state data.
Relationship to REST	While this constraint builds upon Client-Server {307}, it helps enable Cache {310} and Layered System {313}.
Related REST Goals	Modifiability, Scalability, Performance (negative), Visibility, Reliability
Related Service-Orientation Principles	Service Statelessness (298)
Related SOA Patterns	State Messaging [362]

Cache	
Short Definition	*"Service consumers can cache and reuse response message data."*
Long Definition	*"The data provided by a prior response message can be temporarily stored and reused by the service consumer for later request messages."*
Application	• Services must be designed to produce accurate cache control metadata and return it in response messages. Response messages are marked as cacheable or non-cacheable, either with explicit message metadata or as part of the contract definition. • An optional consumer-side or intermediary cache repository enables the consumer to reuse cacheable response data for later request messages. • Request messages must be comparable to determine whether or not they are equivalent. • Contracts must either include explicit statements about the cacheability of responses, or must allow for cache control metadata to be included in responses.
Impacts	• Runtime efficiency is improved by eliminating the need for duplicate response messages to be transmitted and processed. • The cache provides a robust and simple mechanism to perform "lazy replication" of service state data to its consumers. • Some forms of cached data can become stale and outdated if not regularly checked and updated.
Relationship to REST	A number of established techniques for pushing data out to consumers are disallowed by the application of Client-Server {307} and Stateless {308}. The Cache constraint provides a mechanism that is permitted by other constraints and one that results in a simple and robust architecture for reusing and optimizing the distribution of data.
Related REST Goals	Performance, Scalability, Reliability (negative)
Related Service-Orientation Principles	n/a
Related SOA Patterns	State Messaging [362]

Uniform Contract	
Short Definition	*"Service consumers and services share a common, overarching, generic technical contract."*
Long Definition	*"Consumers access service capabilities via methods, media types, and a common resource identifier syntax that are standardized across many consumers and services. Service capabilities provide access to resources that can further provide links to other resources."*
Application	• A uniform contract with generic and reusable methods, media types, and resource identifier syntax is established for a collection of consumers and services. • Consumer message processing logic is designed to be tightly coupled to the uniform contract. • Consumer message processing logic is designed to be decoupled or loosely coupled to service-specific capabilities and resources. • Resources can further provide links to other resources that the service consumer can "discover" and optionally access, dynamically at runtime.
Impacts	• The application of this constraint results in baseline standardization of technical interface characteristics across all services within the scope of application. This level of standardization can foster interoperability across all affected services. • Standardization resulting from Uniform Contract can include canonical schemas associated with media types. The common use of such schemas can further improve the extent of intrinsic interoperability.

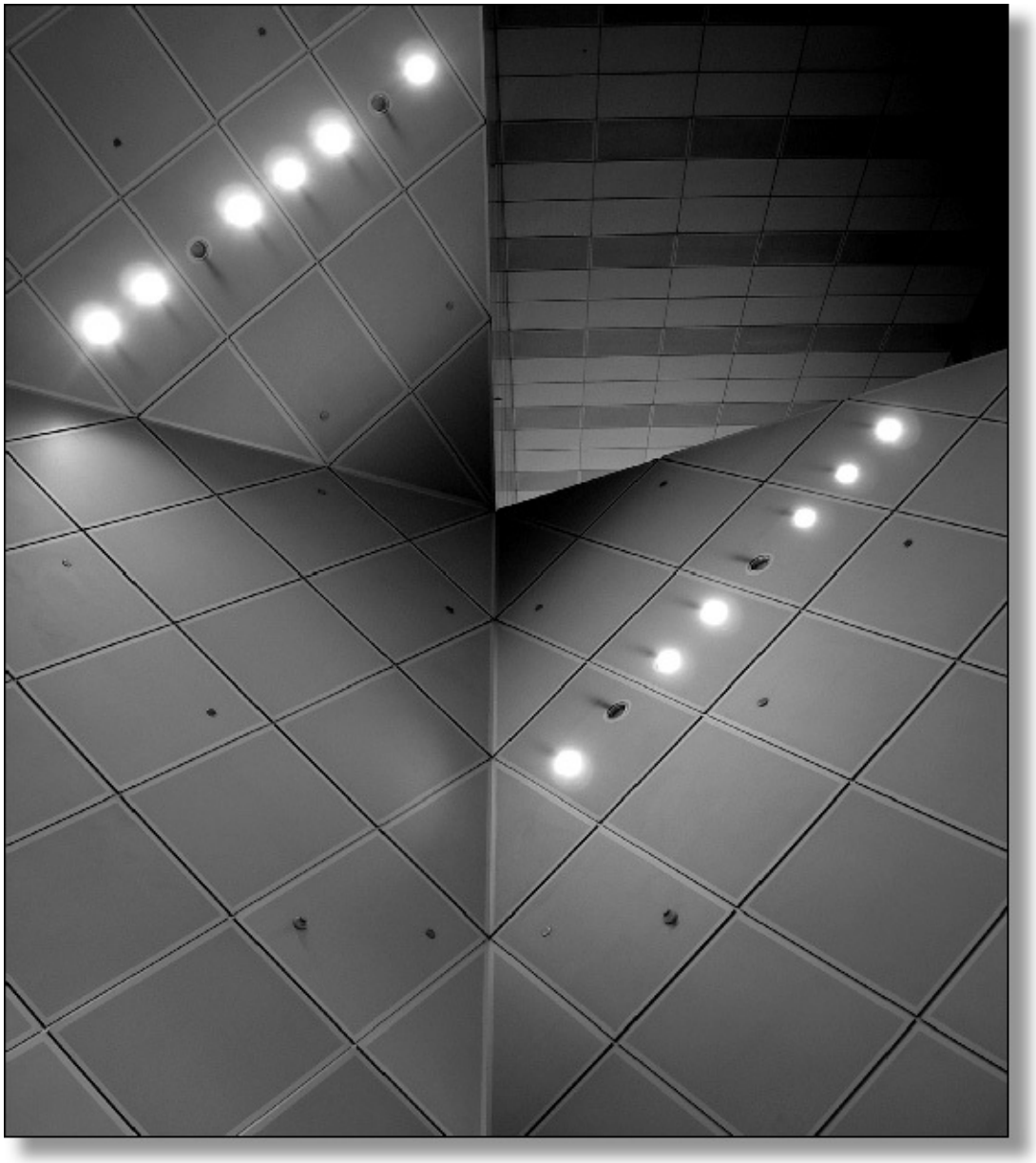
	• By limiting coupling to the uniform contract and leveraging dynamic binding, consumers and services can achieve reduced levels of overall coupling requirements. • It can be difficult to identify and entirely rely on built-in uniform contract semantics for machine-to-machine interactions that need to be reusable by multiple services and their consumers. • Request and response messages based on uniform methods and media types may contain more information than is strictly required for a particular interaction. The transfer of redundant data can increase performance overhead.
Relationship to REST	The Uniform Contract constraint builds upon Client-Server {307} to support reuse and composition of consumers and services.
Related REST Goals	Simplicity, Modifiability, Performance (negative), Visibility
Related Service-Orientation Principles	Standardized Service Contract (291), Service Loose Coupling (293), Service Abstraction (294)
Related SOA Patterns	Decoupled Contract [337]

Layered System	
Short Definition	*"A solution can be comprised of multiple architectural layers."*
Long Definition	*"A solution is defined in terms of architectural layers, where no one layer can see past the next. Layers can be comprised of consumers and services with published contracts or event-driven middleware components (intermediaries) that establish processing layers between consumers and services. In either case, logic within a given solution layer cannot have knowledge beyond the immediate layers above or below it (within the solution hierarchy)."*
Application	• Consumers are designed to invoke services without knowledge of what other services those services may also invoke. • Intermediaries are added to perform runtime message processing without knowledge of how those messages may be further processed beyond the next layer of processing. • The solution architecture is designed to allow new middleware layers to be added or old middleware layers to be removed without changing the technical contract between services and consumers. • Request and response messages must not reveal which layer the message comes from to their recipients.
Impacts	• At the consumer/service level, this constraint ensures an extent of information hiding, which naturally reduces consumer-to-service coupling. • At the middleware component level, this constraint advocates the use of cross-cutting agents capable of performing generic, utility-centric functions on messages exchanged by consumers and services.

	• These types of architectural layers can provide a flexible means of evolving a solution architecture and/or its underlying infrastructure while minimizing the impact on the solution logic itself. • The increased separation and distribution of moving parts performing solution logic processing can negatively impact the overall performance overhead (especially when middleware components are being reused by multiple solutions). • By limiting knowledge of the entire solution architecture to consumer designers, opportunities for optimizing the runtime performance of a solution can be lost.
Relationship to REST	The middleware components commonly introduced by the application of this constraint can directly support or enable Uniform Contract {311}, Cache {310}, and Stateless {308}.
Related REST Goals	Modifiability, Scalability, Performance (negative), Simplicity, Visibility
Related Service-Orientation Principles	Service Loose Coupling (293), Service Abstraction (294)
Related SOA Patterns	Capability Composition [328], Service Agent [357]

Code-on-Demand	
Short Definition	*"Service consumers support the execution of deferred service logic."*
Long Definition	*"Service consumer architectures include an execution environment for logic provided by a service. This deferred logic can be used to extend the functionality of the consumer, or to temporarily specialize it for a particular purpose."*
Application	• Service consumers are designed to process logic offloaded to them by services at runtime. • Services make explicit decisions as to whether they will execute logic themselves or defer the execution of that logic to their consumers.
Impacts	• Features can be dynamically added to consumers without the need for them to be formally upgraded. • Services are able to avoid becoming execution bottlenecks by deferring logic to consumers rather than executing the logic themselves. • The required execution environments for consumers to process service logic can introduce security vulnerabilities.
Relationship to REST	n/a
Related REST Goals	Modifiability, Scalability, Performance, Visibility (negative), Simplicity (negative)
Related Service-Orientation Principles	n/a
Related SOA Patterns	n/a

Appendix C. SOA Design Patterns Reference



This appendix provides profile tables for the patterns referenced throughout this book. As explained in Chapter 1, each pattern reference is suffixed with the page number of its corresponding profile table in this appendix.

What's a Design Pattern?

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

part of a larger collection.

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

Design patterns are helpful because they:

- Represent field-tested solutions to common design problems
- Organize design intelligence into a standardized and easily "referenceable" format
- Are generally repeatable by most IT professionals involved with design
- Can be used to ensure consistency in how systems are designed and built
- Can become the basis for design standards
- Are usually flexible and optional (and openly document the impacts of their application and even suggest alternative approaches)
- Can be used as educational aids by documenting specific aspects of system design (regardless of whether they are applied)
- Can sometimes be applied prior and subsequent to the implementation of a system
- Can be supported via the application of other design patterns that are part of the same collection
- Enrich the vocabulary of a given IT field because each pattern is given a meaningful name

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

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

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

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

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

appropriate.

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

Note

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

What's a Design Pattern Language?

A *pattern language* is a set of related patterns that act as building blocks, in that they can be carried out in one or more predefined or suggested pattern sequences where each subsequent pattern builds upon the former. The notion of a pattern language originated in building architecture as did the term *pattern sequence* used in association with the order in which patterns can be carried out.

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

Structured pattern languages are helpful because they:

- Can organize groups of field-tested design patterns into proposed, field-tested application sequences

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

The SOA Design Patterns book provides an open-ended, master pattern language for SOA. The extent to which different patterns are related can vary, but overall they share a common objective, and endless pattern sequences can be explored.

Pattern Profiles

Every profile table contains the following parts:

- *Requirement* – A requirement is a concise, single-sentence statement that presents the fundamental requirement addressed by the pattern in the form of a question. Every pattern description begins with this statement.
- *Icon* – Each pattern description is accompanied by an icon image that acts as a visual identifier. The icons are displayed together with the requirement statements in each pattern profile.
- *Problem* – The issue causing a problem and the effects of the problem. It is this problem for which the pattern is expected to provide a solution.
- *Solution* – This represents the design solution proposed by the pattern to solve the problem and fulfill the requirement.
- *Application* – This part is dedicated to describing how the pattern can be applied. It can include guidelines, implementation details, and sometimes even a suggested process.
- *Impacts* – This part highlights common consequences, costs, and requirements associated with the application of a pattern and may also provide alternatives that can be considered.
- *Principles* – References to related service-orientation principles.
- *Architecture* – References to related SOA architecture types.

Note that these profile tables provide only summarized versions of the patterns. Complete coverage of SOA design patterns, including case studies, is provided in the *SOA Design Patterns* book.

For more information about this and other titles in the *Prentice Hall Service Technology Series from Thomas Erl*, visit www.servicetechbooks.com. Summarized versions of all SOA pattern profiles can be found online at www.soapatterns.org.

Agnostic Capability

By Thomas Erl

How can multi-purpose service logic be made effectively consumable and composable?

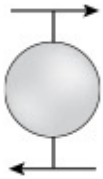
Problem	Service capabilities derived from specific concerns may not be useful to multiple service consumers, thereby reducing the reusability potential of the agnostic service.
Solution	Agnostic service logic is partitioned into a set of well-defined capabilities that address common concerns not specific to any one problem. Through subsequent analysis, the agnostic context of capabilities is further refined.
Application	Service capabilities are defined and iteratively refined through proven analysis and modeling processes.
Impacts	The definition of each service capability requires extra up-front analysis and design effort.
Principles	Standardized Service Contract (291), Service Reusability (295), Service Composability (302)
Architecture	Service

Agnostic Context

By Thomas Erl

How can multi-purpose service logic be positioned as an effective enterprise

resource?

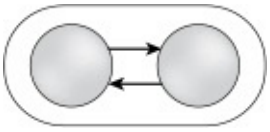


Problem	Multi-purpose logic grouped together with single purpose logic results in programs with little or no reuse potential that introduce waste and redundancy into an enterprise.
Solution	Isolate logic that is not specific to one purpose into separate services with distinct agnostic contexts.
Application	Agnostic service contexts are defined by carrying out service-oriented analysis and service modeling processes.
Impacts	This pattern positions reusable solution logic at an enterprise level, potentially bringing with it increased design complexity and enterprise governance issues.
Principles	Service Reusability (295)
Architecture	Service

Atomic Service Transaction

By Thomas Erl

How can a transaction with rollback capability be propagated across messaging-based services?



Problem	When runtime activities that span multiple services fail, the parent business task is incomplete and actions performed and changes made up to that point may compromise the integrity of the underlying solution and architecture.
Solution	Runtime service activities can be wrapped in a transaction with rollback feature that resets all actions and changes if the parent business task cannot be successfully completed.
Application	A transaction management system is made part of the inventory architecture and then used by those service compositions that require rollback features.
Impacts	Transacted service activities can consume more memory because of the requirement for each service to preserve its original state until it is notified to rollback or commit its changes.
Principles	Service Statelessness (298)
Architecture	Inventory, Composition

Canonical Expression

By Thomas Erl

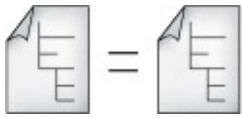
How can service contracts be consistently understood and interpreted?

Problem	Service contracts may express similar capabilities in different ways, leading to inconsistency and risking misinterpretation.
Solution	Service contracts are standardized using naming conventions.
Application	Naming conventions are applied to service contracts as part of formal analysis and design processes.
Impacts	The use of global naming conventions introduces enterprise-wide standards that need to be consistently used and enforced.
Principles	Standardized Service Contract (291), Service Discoverability (300)
Architecture	Enterprise, Inventory, Service

Canonical Schema

By Thomas Erl

How can services be designed to avoid data model transformation?

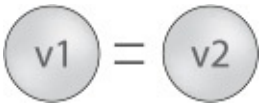


Problem	Services with disparate models for similar data impose transformation requirements that increase development effort, design complexity, and runtime performance overhead.
Solution	Data models for common information sets are standardized across service contracts within an inventory boundary.
Application	Design standards are applied to schemas used by service contracts as part of a formal design process.
Impacts	Maintaining the standardization of contract schemas can introduce significant governance effort and cultural challenges.
Principles	Standardized Service Contract (291)
Architecture	Inventory, Service

Canonical Versioning

By Thomas Erl

How can service contracts within the same service inventory be versioned with minimal impact?

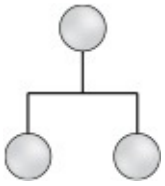


Problem	Service contracts within the same service inventory that are versioned differently will cause numerous interoperability and governance problems.
Solution	Service contract versioning rules and the expression of version information are standardized within a service inventory boundary.
Application	Governance and design standards are required to ensure consistent versioning of service contracts within the inventory boundary.
Impacts	The creation and enforcement of the required versioning standards introduce new governance demands.
Principles	Standardized Service Contract (291)
Architecture	Service, Inventory

Capability Composition

By Thomas Erl

How can a service capability solve a problem that requires logic outside of the service boundary?

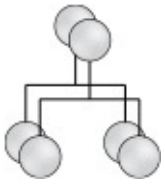


Problem	A capability may not be able to fulfill its processing requirements without adding logic that resides outside of its service's functional context, thereby compromising the integrity of the service context and risking service denormalization.
Solution	When requiring access to logic that falls outside of a service's boundary, capability logic within the service is designed to compose one or more capabilities in other services.
Application	The functionality encapsulated by a capability includes logic that can invoke other capabilities from other services.
Impacts	Carrying out composition logic requires external invocation, which adds performance overhead and decreases service autonomy.
Principles	All
Architecture	Inventory, Composition, Service

Capability Recomposition

By Thomas Erl

How can the same capability be used to help solve multiple problems?

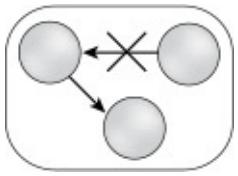


Problem	Using agnostic service logic to only solve a single problem is wasteful and does not leverage the logic's reuse potential.
Solution	Agnostic service capabilities can be designed to be repeatedly invoked in support of multiple compositions that solve multiple problems.
Application	Effective recomposition requires the coordinated, successful, and repeated application of several additional patterns.
Impacts	Repeated service composition demands existing and persistent standardization and governance.
Principles	All
Architecture	Inventory, Composition, Service

Compensating Service Transaction

By Clemens Utschig-Utschig, Berthold Maier, Bernd Trops, Hajo Normann, Torsten Winterberg, Brian Loesgen, Mark Little

How can composition runtime exceptions be consistently accommodated without requiring services to lock resources?

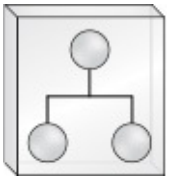


Problem	Whereas uncontrolled runtime exceptions can jeopardize a service composition, wrapping the composition in an atomic transaction can tie up too many resources, thereby negatively affecting performance and scalability.
Solution	Compensating routines are introduced, allowing runtime exceptions to be resolved with the opportunity for reduced resource locking and memory consumption.
Application	Compensation logic is pre-defined and implemented as part of the parent composition controller logic or via individual "undo" service capabilities.
Impacts	Unlike atomic transactions that are governed by specific rules, the use of compensation logic is open-ended and can vary in its actual effectiveness.
Principles	Service Loose Coupling (293)
Architecture	Inventory, Composition

Composition Autonomy

By Thomas Erl

How can compositions be implemented to minimize loss of autonomy?

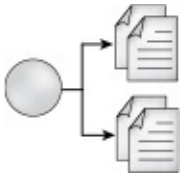


Problem	Composition controller services naturally lose autonomy when delegating processing tasks to composed services, some of which may be shared across multiple compositions.
Solution	All composition participants can be isolated to maximize the autonomy of the composition as a whole.
Application	The agnostic member services of a composition are redundantly implemented in an isolated environment together with the task service.
Impacts	Increasing autonomy on a composition level results in increased infrastructure costs and government responsibilities.
Principles	Service Autonomy (297), Service Reusability (295), Service Composability (302)
Architecture	Composition

Concurrent Contracts

By Thomas Erl

How can a service facilitate multi-consumer coupling requirements and abstraction concerns at the same time?



Problem	A service's contract may not be suitable for or applicable to all potential service consumers.
Solution	Multiple contracts can be created for a single service, each targeted at a specific type of consumer.
Application	This pattern is ideally applied together with Service Façade [360] to support new contracts as required.
Impacts	Each new contract can effectively add a new service endpoint to an inventory, thereby increasing corresponding governance effort.
Principles	Standardized Service Contract (291), Service Loose Coupling (293), Service Reusability (295)
Architecture	Service

Containerization

By Roger Stoffers

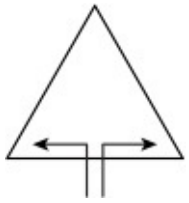
How can an environment be provided with maximum support for services with high-performance recovery and scalability requirements?

Problem	Services deployed on bare metal or virtual servers can impose a significant footprint. Virtualization improves portability but introduces a layer of intermediate processing that can further increase the footprint. Monolithic solution deployments can lead to widespread reduced performance and availability when any one service or solution component suffers an outage or a runtime exception.
Solution	Services are deployed independently, or together with composed services, as autonomous units that are packaged into independently manageable and autonomous container images, each of which includes the services' underlying system dependencies. Tooling is provided to manage the building, deploying and operating of the containers.
Application	A container management system or container engine is used for the deployment and operation of containers.
Impacts	The utilization of containerization technology can impose additional infrastructure requirements, as well as associated increases in the administration overhead of the service architecture.
Principles	Service Autonomy (297), Service Loose Coupling (293)
Architecture	Composition, Service

Content Negotiation

By Raj Balasubramanian, David Booth, Thomas Erl

How can a service capability accommodate service consumers with different data format or representation requirements?

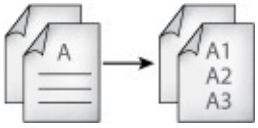


Problem	Different service consumers may have differing requirements for how data provided by a given service capability needs to be formatted or represented.
Solution	Allow the service capability to support alternative formats and representations by providing a means by which consumer and service can "negotiate" data characteristics at runtime.
Application	The pattern is most commonly applied via HTTP media types that can define the format and/or representation of message data. The media type of the data is decoupled from the data itself, allowing the service to support a range of media types. The consumer provides metadata in each request message to identify preferred and supported media types. The service attempts to accommodate preferences, but can also return the data in other supported media types when issuing the response message.
Impacts	Fewer service capabilities are needed to accommodate variation in service consumer requirements. Services are able to support old and new service consumer versions concurrently using the same service capabilities. The complexity of cache implementations is increased, and requires that caching metadata indicate what metadata input to each request may affect which representation will be returned. Requesting metadata that is not abstract enough can introduce consumer to service implementation coupling.
Principles	Standardized Service Contract, (291) Service Loose Coupling (293)
Architecture	Composition, Service

Contract Denormalization

By Thomas Erl

How can a service contract facilitate consumer programs with differing data exchange requirements?

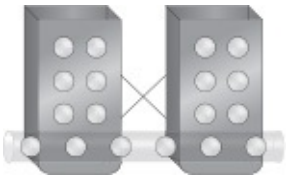


Problem	Services with strictly normalized contracts can impose unnecessary functional and performance demands on some consumer programs.
Solution	Service contracts can include a measured extent of denormalization, allowing multiple capabilities to redundantly express core functions in different ways for different types of consumer programs.
Application	The service contract is carefully extended with additional capabilities that provide functional variations of a primary capability.
Impacts	Overuse of this pattern on the same contract can dramatically increase its size, making it difficult to interpret and unwieldy to govern.
Principles	Standardized Service Contract (291), Service Loose Coupling (293)
Architecture	Service

Cross-Domain Utility Layer

By Thomas Erl

How can redundant utility logic be avoided across domain service inventories?

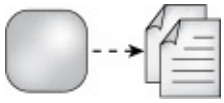


Problem	While domain service inventories may be required for independent business governance, they can impose unnecessary redundancy within utility service layers.
Solution	A common utility service layer can be established, spanning two or more domain service inventories.
Application	A common set of utility services needs to be defined and standardized in coordination with service inventory owners.
Impacts	Increased effort is required to coordinate and govern a cross-inventory utility service layer.
Principles	Service Reusability (295), Service Composability (302)
Architecture	Enterprise, Inventory

Decoupled Contract

By Thomas Erl

How can a service express its capabilities independently of its implementation?

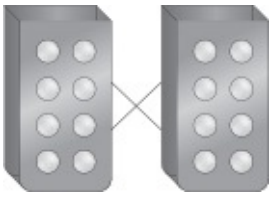


Problem	For a service to be positioned as an effective enterprise resource, it must be equipped with a technical contract that exists independently from its implementation yet still in alignment with other services.
Solution	The service contract is physically decoupled from its implementation.
Application	A service's technical interface is physically separated and subject to relevant service-orientation design principles.
Impacts	Service functionality is limited to the feature-set of the decoupled contract medium.
Principles	Standardized Service Contract (291), Service Loose Coupling (293)
Architecture	Service

Domain Inventory

By Thomas Erl

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



Problem	Establishing a single enterprise service inventory may be unmanageable for some enterprises, and attempts to do so may jeopardize the success of an SOA adoption as a whole.
Solution	Services can be grouped into manageable, domain-specific service inventories, each of which can be independently standardized, governed, and owned.
Application	Inventory domain boundaries need to be carefully established.
Impacts	Standardization disparity between domain service inventories imposes transformation requirements and reduces the overall benefit potential of the SOA adoption.
Principles	Standardized Service Contract (291), Service Abstraction (294), Service Composability (302)
Architecture	Enterprise, Inventory

Dual Protocols

By Thomas Erl

How can a service inventory overcome the limitations of its canonical protocol while still remaining standardized?

Problem	Canonical Protocol requires that all services conform to the use of the same communications technology; however, a single protocol may not be able to accommodate all service requirements, thereby introducing limitations.
Solution	The service inventory architecture is designed to support services based on primary and secondary protocols.
Application	Primary and secondary service levels are created and collectively represent the service endpoint layer. All services are subject to standard service-orientation design considerations and specific guidelines are followed to minimize the impact of not following Canonical Protocol.
Impacts	This pattern can lead to a convoluted inventory architecture, increased governance effort and expense, and (when poorly applied) an unhealthy dependence on Protocol Bridging. Because the endpoint layer is semi-federated, the quantity of potential consumers and reuse opportunities is decreased.
Principles	Standardized Service Contract (291), Service Loose Coupling (293), Service Abstraction (294), Service Autonomy (297), Service Composability (302)
Architecture	Inventory, Service

Enterprise Inventory

By Thomas Erl

How can services be delivered to maximize recomposition?

Problem	Delivering services independently via different project teams across an enterprise establishes a constant risk of producing inconsistent service and architecture implementations, compromising recomposition opportunities.
Solution	Services for multiple solutions can be designed for delivery within a standardized, enterprise-wide inventory architecture wherein they can be freely and repeatedly recomposed.
Application	The enterprise service inventory is ideally modeled in advance, and enterprise-wide standards are applied to services delivered by different project teams.
Impacts	Significant upfront analysis is required to define an enterprise inventory blueprint and numerous organizational impacts result from the subsequent governance requirements.
Principles	Standardized Service Contract (291), Service Abstraction (294), Service Composability (302)
Architecture	Enterprise, Inventory

Entity Abstraction

By Thomas Erl

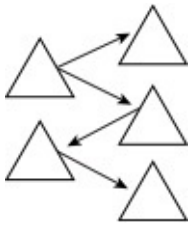
How can agnostic business logic be separated, reused, and governed independently?

Problem	Bundling both process-agnostic and process-specific business logic into the same service eventually results in the creation of redundant agnostic business logic across multiple services.
Solution	An agnostic business service layer can be established, dedicated to services that base their functional context on existing business entities.
Application	Entity service contexts are derived from business entity models and then establish a logical layer that is modeled during the analysis phase.
Impacts	The core, business-centric nature of the services introduced by this pattern require extra modeling and design attention and their governance requirements can impose dramatic organizational changes.
Principles	Service Loose Coupling (293), Service Abstraction (294), Service Reusability (295), Service Composability (302)
Architecture	Inventory, Composition, Service

Entity Linking

By Raj Balasubramanian, David Booth, Thomas Erl

How can services expose the inherent relationships between business entities in order to support loosely-coupled composition?

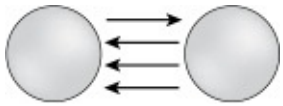


Problem	Business entities have natural relationships, yet entity services are commonly designed autonomously with no indication of these relationships. Service consumers acting as composition controllers are commonly required to have entity linking logic hard-coded in order to work with entity relationships. This limits the composition controller to any additional links that may become relevant and further adds a governance burden to ensure that hard-coded entity linking logic is kept in synch with the business.
Solution	Services inform their consumers about the existence of related entities as part of the consumer's interactions with the services.
Application	Links are included in relevant response messages from the service. Service consumers are able to navigate from entity to entity by following these links, and accumulate further business knowledge along the way. This allows service consumers with little up-front entity linking logic to correctly compose entity services based on their relationships.
Impacts	Resource identifiers representing business entities need to remain relatively stable over the lifespan of the business entities they identify. Once an identifier is known it can be referred to in the future again by the same service consumers. Links can be difficult to define if identifiers for business entities are specific to the services that own them. The application of Lightweight Endpoint can help achieve a uniform syntax for linked identifiers. Links are not valuable if the service consumer is unable to access information about the linked entity. Therefore, the further application of Reusable Contract [355] can ensure that service consumers are able to interact with linked entities.
Principles	Service Reusability (295), Service Abstraction (294), Service Composability (302)
Architecture	Inventory, Service

Event-Driven Messaging

By Mark Little, Thomas Rischbeck, Arnaud Simon

How can service consumers be automatically notified of runtime service events?

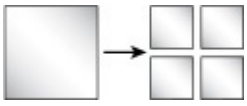


Problem	Events that occur within the functional boundary encapsulated by a service may be of relevance to service consumers, but without resorting to inefficient polling-based interaction, the consumer has no way of learning about these events.
Solution	The consumer establishes itself as a subscriber of the service. The service, in turn, automatically issues notifications of relevant events to this and any of its subscribers.
Application	A messaging framework is implemented capable of supporting the publish-and-subscribe MEP and associated complex event processing and tracking.
Impacts	Event-driven message exchanges cannot easily be incorporated as part of Atomic Service Transaction [324], and publisher/subscriber availability issues can arise.
Principles	Standardized Service Contract (291), Service Loose Coupling (293), Service Autonomy (297)
Architecture	Inventory, Composition

Functional Decomposition

By Thomas Erl

How can a large business problem be solved without having to build a standalone body of solution logic?

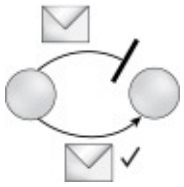


Problem	To solve a large, complex business problem a corresponding amount of solution logic needs to be created, resulting in a self-contained application with traditional governance and reusability constraints.
Solution	The large business problem can be broken down into a set of smaller, related problems, allowing the required solution logic to also be decomposed into a corresponding set of smaller, related solution logic units.
Application	Depending on the nature of the large problem, a service-oriented analysis process can be created to cleanly deconstruct it into smaller problems.
Impacts	The ownership of multiple smaller programs can result in increased design complexity and governance challenges.
Principles	n/a
Architecture	Service

Idempotent Capability

By Cesare Pautasso, Herbjörn Wilhelmsen

How can a service capability safely accept multiple copies of the same message to handle communication failure?



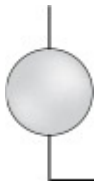
Problem	Network and server hardware failure can lead to lost messages, resulting in cases where a service consumer receives no response to its request. Attempts to reissue the request message can lead to unpredictable or undesirable behavior when the service capability inadvertently receives multiple copies of the same request message.
Solution	Design service capabilities with idempotent logic that enables them to safely accept repeated message exchanges.
Application	Idempotency guarantees that repeated invocations of a service capability are safe and will have no negative effect. Idempotent capabilities are generally limited to read-only data retrieval and queries. For capabilities that do request changes to service state, their logic is generally based on "set," "put" or "delete" actions that have a post-condition that does not depend on the original state of the service. The design of an idempotent capability can include the use of a unique identifier with each request so that repeated requests (with the same identifier value) that have already been processed will be discarded or ignored by the service capability, rather than being processed again.
Impacts	The use of a unique identifier to define an idempotent capability requires session state to be reliably recorded by the service and preserved across server hardware failures. This can harm the scalability of the service, and may be further complicated if redundant service implementations are operating at different sites that experience network failures. Not all service capabilities can be idempotent. Potentially unsafe capabilities include those that need to perform "increment," "reverse" or "escalate" transition functions, where the post-execution condition is dependent upon the original state of the service.
Principles	Standardized Service Contract (291), Service Statelessness (298), Service Composability (302)
Architecture	Inventory, Composition, Service

Inventory Endpoint

By Thomas Erl

How can a service inventory be shielded from external access while still offering

service capabilities to external consumers?

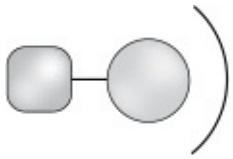


Problem	A group of services delivered for a specific inventory may provide capabilities that are useful to services outside of that inventory. However, for security and governance reasons, it may not be desirable to expose all services or all service capabilities to external consumers.
Solution	Abstract the relevant capabilities into an endpoint service that acts as a the official inventory entry point dedicated to a specific set of external consumers.
Application	The endpoint service can expose a contract with the same capabilities as its underlying services, but augmented with policies or other characteristics to accommodate external consumer interaction requirements.
Impacts	Endpoint services can increase the governance freedom of underlying services but can also increase governance effort by introducing redundant service logic and contracts into an inventory.
Principles	Standardized Service Contract (291), Service Loose Coupling (293), Service Abstraction (294)
Architecture	Inventory

Legacy Wrapper

By Thomas Erl, Satadru Roy

How can wrapper services with non-standard contracts be prevented from spreading indirect consumer-to-implementation coupling?

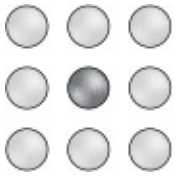


Problem	Wrapper services required to encapsulate legacy logic are often forced to introduce a non-standard service contract with high technology coupling requirements, resulting in a proliferation of implementation coupling throughout all service consumer programs.
Solution	The non-standard wrapper service can be replaced by or further wrapped with a standardized service contract that extracts, encapsulates, and possibly eliminates legacy technical details from the contract.
Application	A custom service contract and required service logic need to be developed to represent the proprietary legacy interface.
Impacts	The introduction of an additional service adds a layer of processing and associated performance overhead.
Principles	Standardized Service Contract (291), Service Loose Coupling (293), Service Abstraction (294)
Architecture	Service

Logic Centralization

By Thomas Erl

How can the misuse of redundant service logic be avoided?



Problem	If agnostic services are not consistently reused, redundant functionality can be delivered in other services, resulting in problems associated with inventory denormalization and service ownership and governance.
Solution	Access to reusable functionality is limited to official agnostic services.
Application	Agnostic services need to be properly designed and governed, and their use must be enforced via enterprise standards.
Impacts	Organizational issues reminiscent of past reuse projects can raise obstacles to applying this pattern.
Principles	Service Reusability (295), Service Composability (302)
Architecture	Inventory, Composition, Service

Microservice Deployment

By Paulo Merson

How can a service be deployed independently to avoid the limitations imposed by a monolithic deployment?

Problem	Services and other components of a software solution are packaged together in a monolithic deployment bundle. Deploying a new version of a service that is part of the solution can require redeploying the entire solution. Also, there is less flexibility to configure service-specific scalability, availability, persistence, monitoring, and security logic.
Solution	Each service is treated as an independent product and is deployed is an isolated package that contributes to service autonomy.
Application	Services are packaged and deployed in a highly autonomous environment that may utilize containerization technology. Packaging and deployment of services are typically highly automated. Services are commonly designed for use with HTTP/REST and to support asynchronous inter-service communication.
Impacts	Services can be developed and evolved more independently. Service deployments can be tailored and new versions can be released with minimal downtime. An increased memory footprint may be required and performance overhead can be imposed due to the increased need for network-based communication.
Principles	Service Autonomy (297), Service Loose Coupling (293)
Architecture	Composition, Service

Note

"Microservice" is an industry term that can be used for services that comply to the micro-service model and to which service-orientation has been applied (and are therefore part of an SOA environment), as well as for services that are not part of an SOA environment. As part of the SOA patterns catalog, the *Microservice Deployment* pattern is authored solely for services that are part of an SOA environment and, most commonly, to which the Micro Task Abstraction [350] pattern has been applied.

Micro Task Abstraction

By Thomas Erl

How can non-agnostic logic with specialized processing requirements be separated and governed independently?

Problem	Grouping non-agnostic logic with specialized processing and deployment requirements together with non-agnostic logic that does not have such requirements can compromise the former's ability to consistently fulfill its requirements.
Solution	Individual units of non-agnostic logic with specialized processing and deployment requirements are separated using the microservice model and abstracted into a microservice layer in which there is the architectural freedom to tailor environments in support of specialized service processing and deployment requirements.
Application	Once non-agnostic business process logic has been separated from agnostic logic, it is reviewed to identify units of logic with specialized processing and deployment requirements suitable for the microservice layer.
Impacts	The abstraction of micro task logic into a separate service layer can introduce analysis, design and governance overhead. The Microservice Deployment [349] pattern is commonly applied to micro task logic in order to realize the necessary service deployment environment. This can introduce disparate communication protocols and further demand specialized implementation technology that may impose new infrastructure, administration and governance requirements.
Principles	Service Abstraction (294), Service Autonomy (297), Service Composability (302), Service Loose Coupling (293)
Architecture	Composition, Inventory, Service

Non-Agnostic Context

By Thomas Erl

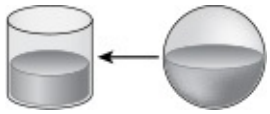
How can single-purpose service logic be positioned as an effective enterprise resource?

Problem	Non-agnostic logic that is not service-oriented can inhibit the effectiveness of service compositions that utilize agnostic services.
Solution	Non-agnostic solution logic suitable for service encapsulation can be located within services that reside as official members of a service inventory.
Application	A single-purpose functional service context is defined.
Impacts	Although they are not expected to provide reuse potential, non-agnostic services are still subject to the rigor of service-orientation.
Principles	Standardized Service Contract (291), Service Composability (302)
Architecture	Service

Partial State Deferral

By Thomas Erl

How can services be designed to optimize resource consumption while still remaining stateful?



Problem	Service capabilities may be required to store and manage large amounts of state data, resulting in increased memory consumption and reduced scalability.
Solution	Even when services are required to remain stateful, a subset of their state data can be temporarily deferred.
Application	Various state management deferral options exist, depending on the surrounding architecture.
Impacts	Partial state management deferral can add to design complexity and bind a service to the architecture.
Principles	Service Statelessness (298)
Architecture	Inventory, Service

Process Abstraction

By Thomas Erl

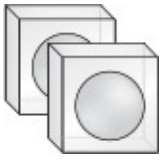
How can non-agnostic process logic be separated and governed independently?

Problem	Grouping task-centric logic together with task-agnostic logic hinders the governance of the task-specific logic and the reuse of the agnostic logic.
Solution	A dedicated parent business process service layer is established to support governance independence and the positioning of task services as potential enterprise resources.
Application	Business process logic is typically filtered out after utility and entity services have been defined, allowing for the definition of task services that comprise this layer.
Impacts	In addition to the modeling and design considerations associated with creating task services, abstracting parent business process logic establishes an inherent dependency on carrying out that logic via the composition of other services.
Principles	Service Loose Coupling (293), Service Abstraction (294), Service Composability (302)
Architecture	Inventory, Composition, Service

Redundant Implementation

By Thomas Erl

How can the reliability and availability of a service be increased?

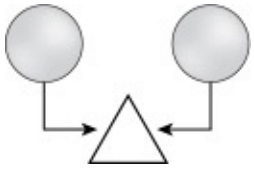


Problem	A service that is being actively reused introduces a potential single point of failure that may jeopardize the reliability of all compositions in which it participates if an unexpected error condition occurs.
Solution	Reusable services can be deployed via redundant implementations or with failover support.
Application	The same service implementation is redundantly deployed or supported by infrastructure with redundancy features.
Impacts	Extra governance effort is required to keep all redundant implementations in synch.
Principles	Service Autonomy (297)
Architecture	Service

Reusable Contract

By Raj Balasubramanian, Benjamin Carlyle, Thomas Erl, Cesare Pautasso

How can service consumers compose services without having to couple themselves to service-specific contracts?

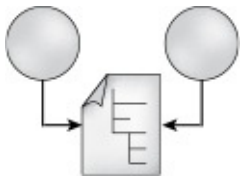


Problem	To access a service capability of a service with a service-specific contract, the service consumer must be designed to couple itself to the service contract. When the service contract changes, the service consumer may no longer be functional. To access a new version of the service contract, or to access other service contracts in order to compose other services, the service consumer must be subjected to additional development cycles, thereby incurring time, effort, and expense.
Solution	Limit tight coupling to a common, reusable technical contract that is shared by multiple services. The technical contract provides only generic, high-level functions that are less likely to be impacted when service logic changes.
Application	A reusable service contract can provide abstract and agnostic data exchange methods, none of which are related to a specific business function. Methods within a reusable contract are typically focused on types of data rather than on the business context of the data. The set of methods of the reusable contract is complemented by service-specific resource identifiers and media types to apply the context established by reusable methods to individual service capabilities. HTTP provides a reusable contract via generic methods, such as GET, PUT, and DELETE, that allow consumer programs to access Web-based resources by further providing resource identifiers. The combination of the resource identifier and the HTTP method and media type can comprise a service-specific capability. A reusable contract can also be created using a centralized WSDL definition, as long as the operations defined are sufficiently generic.
Impacts	Sharing the same contract across services increases the importance of getting the contract right, both initially, and over the contract's lifetime. The reusable contract may still need to change if new services with new high-level functional requirements are introduced into the service inventory. The reusable contract can lack sufficient metadata to effectively enable a service to be discovered. Service-specific metadata may need to be maintained separately from the reusable contract definition to ensure that service consumers are able to select the correct service capability with which to interact.
Principles	Standardized Service Contract (291), Service Loose Coupling (293), Service Abstraction (294), Service Discoverability (300), Service Composability (302)
Architecture	Inventory, Composition, Service

Schema Centralization

By Thomas Erl

How can service contracts be designed to avoid redundant data representation?



Problem	Different service contracts often need to express capabilities that process similar business documents or data sets, resulting in redundant schema content that is difficult to govern.
Solution	Select schemas that exist as physically separate parts of the service contract are shared across multiple contracts.
Application	Up-front analysis effort is required to establish a schema layer independent of and in support of the service layer.
Impacts	Governance of shared schemas becomes increasingly important as multiple services can form dependencies on the same schema definitions.
Principles	Standardized Service Contract (291), Service Loose Coupling (293)
Architecture	Inventory, Service

Service Agent

By Thomas Erl

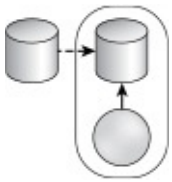
How can event-driven logic be separated and governed independently?

Problem	Service compositions can become large and inefficient, especially when required to invoke granular capabilities across multiple services.
Solution	Event-driven logic can be deferred to event-driven programs that don't require explicit invocation, thereby reducing the size and performance strain of service compositions.
Application	Service agents can be designed to automatically respond to predefined conditions without invocation via a published contract.
Impacts	The complexity of composition logic increases when it is distributed across services, and event-driven agents and reliance on service agents can further tie an inventory architecture to proprietary vendor technology.
Principles	Service Loose Coupling (293), Service Reusability (295)
Architecture	Inventory, Composition

Service Data Replication

By Thomas Erl

How can service autonomy be preserved when services require access to shared data sources?

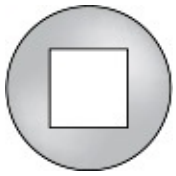


Problem	Service logic can be deployed in isolation to increase service autonomy, but services continue to lose autonomy when requiring access to shared data sources.
Solution	Services can have their own dedicated databases with replication to shared data sources.
Application	An additional database needs to be provided for the service and one or more replication channels need to be enabled between it and the shared data sources.
Impacts	This pattern results in additional infrastructure cost and demands, and an excess of replication channels can be difficult to manage.
Principles	Service Autonomy (297)
Architecture	Inventory, Service

Service Encapsulation

By Thomas Erl

How can solution logic be made available as a resource of the enterprise?

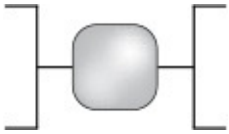


Problem	Solution logic designed for a single application environment is typically limited in its potential to interoperate with or be leveraged by other parts of an enterprise.
Solution	Solution logic can be encapsulated by a service so that it is positioned as an enterprise resource capable of functioning beyond the boundary for which it is initially delivered.
Application	Solution logic suitable for service encapsulation needs to be identified.
Impacts	Service-encapsulated solution logic is subject to additional design and governance considerations.
Principles	n/a
Architecture	Service

Service Façade

By Thomas Erl

How can a service accommodate changes to its contract or implementation while allowing the core service logic to evolve independently?



Problem	The coupling of the core service logic to contracts and implementation resources can inhibit its evolution and negatively impact service consumers.
Solution	A service façade component is used to abstract a part of the service architecture with negative coupling potential.
Application	A separate façade component is incorporated into the service design.
Impacts	The addition of the façade component introduces design effort and performance overhead.
Principles	Standardized Service Contract (291), Service Loose Coupling (293)
Architecture	Service

Service Normalization

By Thomas Erl

How can a service inventory avoid redundant service logic?

Problem	When delivering services as part of a service inventory, there is a constant risk that services will be created with overlapping functional boundaries, making it difficult to enable wide-spread reuse.
Solution	The service inventory needs to be designed with an emphasis on service boundary alignment.
Application	Functional service boundaries are modeled as part of a formal analysis process and persist throughout inventory design and governance.
Impacts	Ensuring that service boundaries are and remain well-aligned introduces extra up-front analysis and on-going governance effort.
Principles	Service Autonomy (297)
Architecture	Inventory, Service

State Messaging

By Anish Karmarkar

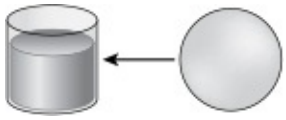
How can a service remain stateless while participating in stateful interactions?

Problem	When services are required to maintain state information in memory between message exchanges with consumers, their scalability can be comprised, and they can become a performance burden on the surrounding infrastructure.
Solution	Instead of retaining the state data in memory, its storage is temporarily delegated to messages.
Application	Depending on how this pattern is applied, both services and consumers may need to be designed to process message-based state data.
Impacts	This pattern may not be suitable for all forms of state data, and should messages be lost, any state information they carried may be lost as well.
Principles	Standardized Service Contract (201), Service Statelessness (298), Service Composability (302)
Architecture	Composition, Service

State Repository

By Thomas Erl

How can service state data be persisted for extended periods without consuming service runtime resources?



Problem	Large amounts of state data cached to support the activity within a running service composition can consume too much memory, especially for long-running activities, thereby decreasing scalability.
Solution	State data can be temporarily written to and then later retrieved from a dedicated state repository.
Application	A shared or dedicated repository is made available as part of the inventory or service architecture.
Impacts	The addition of required write and read functionality increases the service design complexity and can negatively affect performance.
Principles	Service Statelessness (298)
Architecture	Inventory, Service

Utility Abstraction

By Thomas Erl

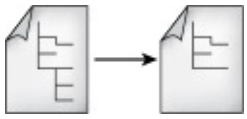
How can common non-business centric logic be separated, reused, and independently governed?

Problem	When non-business centric processing logic is packaged together with business-specific logic, it results in the redundant implementation of common utility functions across different services.
Solution	A service layer dedicated to utility processing is established, providing reusable utility services for use by other services in the inventory.
Application	The utility service model is incorporated into analysis and design processes in support of utility logic abstraction, and further steps are taken to define balanced service contexts.
Impacts	When utility logic is distributed across multiple services it can increase the size, complexity, and performance demands of compositions.
Principles	Service Loose Coupling (293), Service Abstraction (294), Service Reusability (295), Service Composability (302)
Architecture	Inventory, Composition, Service

Validation Abstraction

By Thomas Erl

How can service contracts be designed to more easily adapt to validation logic changes?

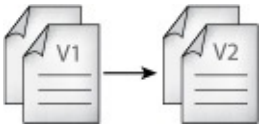


Problem	Service contracts that contain detailed validation constraints become more easily invalidated when the rules behind those constraints change.
Solution	Granular validation logic and rules can be abstracted away from the service contract, thereby decreasing constraint granularity and increasing the contract's potential longevity.
Application	Abstracted validation logic and rules need to be moved to the underlying service logic, a different service, a service agent, or elsewhere.
Impacts	This pattern can somewhat decentralize validation logic and can also complicate schema standardization.
Principles	Standardized Service Contract (291), Service Loose Coupling (293), Service Abstraction (294)
Architecture	Service

Version Identification

By David Orchard, Chris Riley

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



Problem	When an already-published service contract is changed, unaware consumers will miss the opportunity to leverage the change or may be negatively impacted by the change.
Solution	Versioning information pertaining to compatible and incompatible changes can be expressed as part of the service contract, both for communication and enforcement purposes.
Application	With Web service contracts, version numbers can be incorporated into namespace values and as annotations.
Impacts	This pattern may require that version information be expressed with a proprietary vocabulary that needs to be understood by consumer designers in advance.
Principles	Standardized Service Contract (291)
Architecture	Service

Appendix D. The Annotated SOA Manifesto

The SOA Manifesto

The SOA Manifesto Explored

The SOA Manifesto is a formal declaration that explains the underlying design philosophy of SOA and service-orientation. Authored by a working group comprised of industry thought leaders, the SOA Manifesto addresses the core values and priorities of service-orientation. By studying the SOA Manifesto we can gain valuable perspectives and insights into the service-orientation design

paradigm.

This appendix first presents the SOA Manifesto and then breaks it down to elaborate on the meanings and implications of its individual statements. In addition to fostering a deeper understanding of service-orientation, this exploration of values and priorities can help determine their compatibility with an organization's own values, priorities, and goals.

The SOA Manifesto

The following is the verbatim SOA Manifesto, as originally published at www.soa-manifesto.org.

Service orientation is a paradigm that frames what you do. Service-oriented architecture (SOA) is a type of architecture that results from applying service orientation.

We have been applying service orientation to help organizations consistently deliver sustainable business value, with increased agility and cost effectiveness, in line with changing business needs.

Through our work we have come to prioritize:

- *Business value over technical strategy*
- *Strategic goals over project-specific benefits*
- *Intrinsic interoperability over custom integration*
- *Shared services over specific-purpose implementations*
- *Flexibility over optimization*
- *Evolutionary refinement over pursuit of initial perfection*

That is, while we value the items on the right, we value the items on the left more.

Guiding Principles

We follow these principles:

- *Respect the social and power structure of the organization.*
- *Recognize that SOA ultimately demands change on many levels.*
- *The scope of SOA adoption can vary. Keep efforts manageable and within meaningful boundaries.*
- *Products and standards alone will neither give you SOA nor apply the service orientation paradigm for you.*
- *SOA can be realized through a variety of technologies and standards.*

- *Establish a uniform set of enterprise standards and policies based on industry, de facto, and community standards.*
- *Pursue uniformity on the outside while allowing diversity on the inside.*
- *Identify services through collaboration with business and technology stakeholders.*
- *Maximize service usage by considering the current and future scope of utilization.*
- *Verify that services satisfy business requirements and goals.*
- *Evolve services and their organization in response to real use.*
- *Separate the different aspects of a system that change at different rates.*
- *Reduce implicit dependencies and publish all external dependencies to increase robustness and reduce the impact of change.*
- *At every level of abstraction, organize each service around a cohesive and manageable unit of functionality.*

The SOA Manifesto Explored

Subsequent to the announcement of the SOA Manifesto, an annotated version was authored specifically for the *Next Generation SOA: A Concise Introduction to Service Technology & Service-Orientation* book. It was published in advance at www.soa-manifesto.com to facilitate discussion of the manifesto's statements within the industry. Provided in this section is the original Annotated SOA Manifesto content with some minor revisions.

Preamble

Service orientation is a paradigm that frames what you do. Service-oriented architecture (SOA) is a type of architecture that results from applying service orientation.

From the beginning it was understood that this was to be a manifesto about two distinct yet closely related topics: the service-oriented architectural model and service orientation, the paradigm through which the architecture is defined. The format of this manifesto was modeled after the Agile Manifesto, which limits content to concise statements that express ambitions, values, and guiding principles for realizing those ambitions and values. Such a manifesto is not a specification, a reference model, or even a white paper, and without an option to provide actual definitions, we decided to add this preamble in order to clarify how and why these terms are referenced in other parts of the manifesto

document.

We have been applying service orientation.…

The service orientation paradigm is best viewed as a method or an approach for realizing a specific target state that is further defined by a set of strategic goals and benefits. When we apply service orientation, we shape software programs and technology architecture in support of realizing this target state. This is what qualifies technology architecture as being service-oriented.

…to help organizations consistently deliver sustainable business value, with increased agility and cost effectiveness.…

This continuation of the preamble highlights some of the most prominent and commonly expected strategic benefits of service-oriented computing. Understanding these benefits helps shed some light on the aforementioned target state we intend to realize as a result of applying service-orientation.

Agility at a business level is comparable to an organization's responsiveness. The more easily and effectively an organization can respond to business change, the more efficient and successful it will be at adapting to the impacts of the change (and further leveraging whatever benefits the change may bring about).

Service-orientation positions services as IT assets that are expected to provide repeated value over time that far exceeds the initial investment required for their delivery. Cost-effectiveness relates primarily to this expected return on investment. In many ways, an increase in cost-effectiveness goes hand-in-hand with an increase in agility; if there is more opportunity to reuse existing services, then there is generally less expense required to build new solutions.

"Sustainable" business value refers to the long-term goals of service-orientation to establish software programs as services that possess the inherent flexibility to be continually composed into new solution configurations and evolved to accommodate ever-changing business requirements.

…in line with changing business needs.

These last six words of the preamble are key to understanding the underlying philosophy of service-oriented computing. The need to accommodate business change on an ongoing basis is foundational to service-orientation and considered a fundamental overarching strategic goal.

Priorities

Through our work we have come to prioritize:

The upcoming statements establish a core set of values, each of which is

expressed as a prioritization over something that is also considered of value. The intent of this value system is to address the hard choices that need to be made on a regular basis in order for the strategic goals and benefits of service-oriented computing to be consistently realized.

Business value over technical strategy

As stated previously, the need to accommodate business change is an overarching strategic goal. Therefore, the foundational quality of service-oriented architecture and of any software programs, solutions, and ecosystems that result from the adoption of service-orientation is that they are business-driven. It is not about technology determining the direction of the business; it is about the business vision dictating the utilization of technology.

This priority can have a profound ripple effect within the regions of an IT enterprise. It introduces changes to just about all parts of IT delivery lifecycles, from how we plan for and fund automation solutions to how we build and govern them. All other values and principles in the manifesto, in one way or another, support the realization of this value.

Strategic goals over project-specific benefits

Historically, many IT projects focused solely on building applications designed specifically to automate business process requirements that were current at that time. This fulfilled immediate (tactical) needs, but as more of these single-purpose applications were delivered, it resulted in an IT enterprise filled with islands of logic and data referred to as application “silos.” As new business requirements would emerge, either new silos were created or integration channels between silos were established. As yet more business change arose, integration channels had to be augmented, even more silos had to be created, and soon the IT enterprise landscape became convoluted and increasingly burdensome, expensive, and slow to evolve.

In many ways, service-orientation emerged in response to these problems. It is a paradigm that provides an alternative to project-specific, silo-based, and integrated application development by adamantly prioritizing the attainment of long-term, strategic business goals. The target state advocated by service-orientation does not have traditional application silos. And even when legacy resources and application silos exist in environments where service-orientation is adopted, the target state is one where they are harmonized to whatever extent feasible.

Intrinsic interoperability over custom integration

For software programs to share data they need to be interoperable. If software

programs are not designed to be compatible, they will likely not be interoperable. To enable interoperability between incompatible software programs requires that they be integrated. Integration is therefore the effort required to achieve interoperability between disparate software programs.

Although often necessary, customized integration can be expensive and time-consuming and can lead to fragile architectures that are burdensome to evolve. One of the goals of service-orientation is to minimize the need for customized integration by shaping software programs (within a given domain) so that they are natively compatible. This is a quality referred to as intrinsic interoperability. The design principles encompassed by the service-orientation paradigm are geared toward establishing intrinsic interoperability on several levels.

Intrinsic interoperability, as a characteristic of software programs that reside within a given domain, is key to realizing strategic benefits, such as increased cost-effectiveness and agility.

Shared services over specific-purpose implementations

When applied to a meaningful extent, service-orientation principles shape a software program into a unit of service-oriented logic that can be legitimately referred to as a service.

Services are equipped with concrete characteristics (such as those that enable intrinsic interoperability) that directly support the previously described target state. One of these characteristics, fostered specifically by the application of the Service Reusability (295) principle, is the encapsulation of multi-purpose logic that can be shared and reused in support of the automation of different business processes.

A shared service establishes itself as an IT asset that can provide repeated business value while decreasing the expense and effort to deliver new automation solutions. While there is value in traditional, single-purpose applications that solve tactical business requirements, the use of shared services provides greater value in realizing the strategic goals of service-oriented computing (which again includes an increase in cost-effectiveness and agility).

Flexibility over optimization

This is perhaps the broadest of the value prioritization statements and is best viewed as a guiding philosophy for how to better prioritize various considerations when delivering and evolving individual services and inventories of services.

Optimization primarily refers to the fulfillment of tactical gains by tuning a given application design or expediting its delivery to meet immediate needs.

There is nothing undesirable about this, except that it can lead to the aforementioned silo-based environments when not properly prioritized in relation to fostering flexibility.

For example, the characteristic of flexibility goes beyond the ability for services to effectively (and intrinsically) share data. To be truly responsive to ever-changing business requirements, services must also be flexible in how they can be combined and aggregated into composite solutions. Unlike traditional distributed applications that often were relatively static despite the fact that they were componentized, service compositions need be designed with a level of inherent flexibility that allows for constant augmentation. This means that when an existing business process changes or when a new business process is introduced, we need to be able to add, remove, and extend services within the composition architecture with minimal (integration) effort. This is why Service Composability (302) is one of the key service-orientation design principles.

Evolutionary refinement over pursuit of initial perfection

There is a common point of confusion when it comes to the term “agility” in relation to service-orientation. Some design approaches advocate the rapid delivery of software programs for immediate gains. This can be considered “tactical agility,” as the focus is on tactical, short-term benefit. Service-orientation advocates the attainment of agility on an organizational or business level with the intention of empowering the organization, as a whole, to be responsive to change. This form of organizational agility can also be referred to as “strategic agility” because the emphasis is on longevity in that, with every software program we deliver, we want to work toward a target state that fosters agility with long-term strategic value.

For an IT enterprise to enable organizational agility, it must evolve in tandem with the business. We generally cannot predict how a business will need to evolve over time and therefore we cannot initially build the perfect services. At the same time, there is usually a wealth of knowledge already present within an organization’s existing business intelligence that can be harvested during the analysis and modeling stages of SOA projects.

This information, together with service-orientation principles and proven methodologies, can help us identify and define a set of services that capture how the business exists and operates today while being sufficiently flexible to adapt to how the business changes over time.

That is, while we value the items on the right, we value the items on the left more.

By studying how these values are prioritized, we gain insight into what distinguishes service-orientation from other design approaches and paradigms. In addition to establishing fundamental criteria that we can use to determine how compatible service-orientation is for a given organization, it can further help determine the extent to which service-orientation can or should be adopted.

An appreciation of the core values can also help us understand how challenging it may be to successfully carry out SOA projects within certain environments. For example, several of these prioritizations may clash head-on with established beliefs and preferences. In such a case, the benefits of service-orientation need to be weighed against the effort and impact their adoption may have (not just on technology, but also on the organization and IT culture).

The upcoming guiding principles were provided to help address many of these types of challenges.

Guiding Principles

We follow these principles:

So far, the manifesto has established an overall vision as well as a set of core values associated with the vision. The remainder of the declaration is comprised of a set of principles that are provided as guidance for adhering to the values and realizing the vision.

It's important to keep in mind that these are *guiding* principles that were authored specifically in support of this manifesto. They are not to be confused with the *design* principles that comprise service-orientation.

Respect the social and power structure of the organization.

One of the most common SOA pitfalls is approaching adoption as a technology-centric initiative. Doing so almost always leads to failure because we are simply not prepared for the inevitable organizational impacts.

The adoption of service-orientation is about transforming the way we automate business. However, regardless of what plans we may have for making this transformation effort happen, we must always begin with an understanding and an appreciation of the organization, its structure, its goals, and its culture.

The adoption of service-orientation is very much a human experience. It requires support from those in authority and asks that the IT culture adopt a strategic, community-centric mindset. We must fully acknowledge and plan for this level of organizational change in order to receive the necessary long-term commitments required to achieve the target state of service-orientation.

These types of considerations not only help us determine how to best proceed with an SOA initiative, they further assist us in defining the most appropriate scope and approach for adoption.

Recognize that SOA ultimately demands change on many levels.

There's a saying that goes: "Success is being prepared for opportunity." Perhaps the number one lesson learned from SOA projects that have been carried out in the past is that we must fully comprehend and then plan and prepare for the volume and range of change that is brought about as a result of adopting service-orientation. Here are some examples.

Service-orientation changes how we build automation solutions by positioning software programs as IT assets with long-term, repeatable business value. Depending on the extent to which cloud-based infrastructure may be leveraged, a significant up-front investment may be required to create an environment comprised of such assets. Furthermore, an ongoing commitment is required to maintain and leverage their value. So, right out of the gate, changes are required to how we fund, measure, and maintain systems within the IT enterprise.

Additionally, because service-orientation introduces services that are positioned as resources of the enterprise, there will be changes in how we own different parts of systems and regulate their design and usage, not to mention changes to the infrastructure required to guarantee continuous scalability and reliability. Mature SOA governance systems and the service technologies can address these concerns.

The scope of SOA adoption can vary. Keep efforts manageable and within meaningful boundaries.

A common myth has been that in order to realize the strategic goals of service-oriented computing, service-orientation must be adopted on an enterprise-wide basis. This means establishing and enforcing design and industry standards across the IT enterprise so as to create an enterprise-wide inventory of intrinsically interoperable services. While there is nothing wrong with this ideal, it is not a realistic goal for many organizations, especially those with larger IT enterprises.

The most appropriate scope for any given SOA adoption effort needs to be determined as a result of planning and analysis in conjunction with pragmatic considerations, such as the aforementioned impacts on organizational structures, areas of authority, and cultural changes that are brought about. Taking the Balanced Scope pillar into account during the planning stages assists in determining a suitable, initial adoption scope based on an organization's maturity

and readiness.

These factors further help determine a scope of adoption that is deemed manageable. But for any adoption effort to result in an environment that progresses the IT enterprise toward the desired strategic target state, the scope must also be meaningful. In other words, it must be meaningfully cross-silo so that collections of services can be delivered in relation to each other within a pre-defined boundary. In other words, we want to create "continents of services," not the dreaded "islands of services."

This concept of building independently owned and governed service inventories within domains of the same IT enterprise is based on the Domain Inventory [338] design pattern that was originally published as part of the SOA design patterns catalog at www.soapatterns.org. This approach reduces many of the risks that are commonly attributed to "big-bang" SOA projects and furthermore mitigates the impact of both organizational and technological changes (because the impact is limited to a segmented and managed scope). It is also an approach that allows for phased adoption where one domain service inventory can be established at a time.

Products and standards alone will neither give you SOA nor apply the service-orientation paradigm for you.

This guiding principle addresses two separate but very much related myths. The first is that you can buy your way into SOA with modern technology products, and the second is the assumption that the adoption of industry standards (such as XML, WSDL, SCA, etc.) will naturally result in service-oriented technology architecture.

The vendor and industry standards communities have been credited with building modern service technology innovation upon non-proprietary frameworks and platforms. Everything from service virtualization to cloud computing and grid computing has helped advance the potential for building sophisticated and complex service-oriented solutions. However, none of these technologies are exclusive to SOA. You can just as easily build silo-based systems in the cloud as you can on your own private servers.

There is no such thing as "SOA in a box" because in order to achieve service-oriented technology architecture, service-orientation needs to be successfully applied; this, in turn, requires everything that we design and build to be driven by the unique direction, vision, and requirements of the business.

SOA can be realized through a variety of technologies and standards.

Service-orientation is a technology-neutral and vendor-neutral paradigm.

Service-oriented architecture is a technology-neutral and vendor-neutral architectural model. Service-oriented computing can be viewed as a specialized form of distributed computing. Service-oriented solutions can therefore be built using just about any technologies and industry standards suitable for distributed computing.

While some technologies (especially those based on industry standards) can increase the potential of applying some service-orientation design principles, it is really the potential to fulfill business requirements that ultimately determines the most suitable choice of technologies and industry standards. SOA design patterns, such as Dual Protocols [339] and Concurrent Contracts [332], support the use and standardization of alternative service technologies within the same service inventory.

Establish a uniform set of enterprise standards and policies based on industry, de facto, and community standards.

Industry standards represent non-proprietary technology specifications that help establish, among other things, consistent baseline characteristics (such as transport, interface, message format, etc.) of technology architecture. However, the use of industry standards alone does not guarantee that services will be intrinsically interoperable.

For two software programs to be fully compatible, additional conventions (such as data models and policies) need to be adhered to. This is why IT enterprises must establish and enforce design standards. Failure to properly standardize and regulate the standardization of services within a given domain will begin to tear at the fabric of interoperability upon which the realization of many strategic benefits relies.

This guiding principle advocates the use of enterprise design standards and design principles, such as Standardized Service Contract (291) and Service Loose Coupling (293). It also reminds us that, whenever possible and feasible, custom design standards should be based upon and incorporate standards and service-orientation design principles already in use by the industry and the community in general.

Pursue uniformity on the outside while allowing diversity on the inside.

Federation can be defined as the unification of a set of disparate entities. While allowing each entity to be independently governed on the inside, all agree to adhere to a common, unified front.

A fundamental part of service-oriented architecture is the introduction of a federated endpoint layer that abstracts service implementation details while

publishing a set of endpoints that represent individual services within a given domain in a unified manner. Accomplishing this generally involves achieving unity based on a combination of industry and design standards. The consistency of this unity across services is key to realizing intrinsic interoperability, as it represents the primary purpose and responsibility of the Standardized Service Contract (291) design principle.

A federated endpoint layer further helps increase opportunities to explore vendor-diversity options. For example, one service may need to be built upon a completely different platform than another. As long as these services maintain compatible endpoints, the governance of their respective implementations can remain independent. This not only highlights that services can be built using different implementation mediums (such as EJB, .NET, SOAP, REST, etc.), it also emphasizes that different intermediary platforms and technologies can be utilized together, as required.

Note that this type of diversity comes with a price. This principle does not advocate diversification itself—it simply recommends that we allow diversification when justified, so that "best-of-breed" technologies and platforms can be leveraged to maximize business requirements fulfillment.

Identify services through collaboration with business and technology stakeholders.

In order for technology solutions to be business-driven, the technology must be in sync with the business. Therefore, another goal of service-oriented computing is to align technology and business via the application of service-orientation. The stage at which this alignment is initially accomplished is during the analysis and modeling processes that usually precede actual service development and delivery.

The critical ingredient to carrying out service-oriented analysis is to have both business and technology experts working hand-in-hand to identify and define candidate services. For example, business experts can help accurately define functional contexts pertaining to business-centric services, while technology experts can provide pragmatic input to ensure that the granularity and definition of conceptual services remains realistic in relation to their eventual implementation environments.

Maximize service usage by considering the current and future scope of utilization.

The extent of a given SOA project may be enterprise-wide or may be limited to a domain of the enterprise. Whatever the scope, a pre-defined boundary is

established to encompass an inventory of services that need to be conceptually modeled before they can be developed. By modeling multiple services in relation to each other, we essentially establish a blueprint of the services we will eventually be building. This exercise is critical when attempting to identify and define services that can be shared by different solutions.

There are various methodologies and approaches that can be used to carry out service-oriented analysis stages. However, a common thread among all of them is that the functional boundaries of services be normalized to avoid redundancy. Even then, normalized services do not necessarily make for highly reusable services. Other factors come into play, such as service granularity, autonomy, state management, scalability, composability, and the extent to which service logic is sufficiently generic so that it can be effectively reused.

These types of considerations as guided by business and technology expertise provide the opportunity to define services that capture current utilization requirements while possessing the flexibility to adapt to future change.

Verify that services satisfy business requirements and goals.

As with anything, services can be misused. When growing and managing a portfolio of services, their usage and effectiveness at fulfilling business requirements need to be verified and measured. Modern tools provide various means of monitoring service usage, but there are intangibles that also need to be taken into consideration to ensure that services are not just used because they are available, but to verify that they are truly fulfilling business needs and meeting expectations.

This is especially true with shared services that shoulder multiple dependencies. Not only do shared services require adequate infrastructure to guarantee scalability and reliability for all of the solutions that reuse them, they also need to be designed and extended with great care to ensure their functional contexts are never skewed.

Evolve services and their organization in response to real use.

This guiding principle ties directly back to the “Evolutionary refinement over pursuit of initial perfection” value statement, as well as the overall goal of maintaining an alignment of business and technology.

We can never expect to rely on guesswork when it comes to determining service granularity, the range of functions that services need to perform, or how services will need to be organized into compositions. Based on whatever extent of analysis we are able to initially perform, a given service will be assigned a defined functional context and will contain one or more functional capabilities

that likely involve it in one or more service compositions.

As real-world business requirements and circumstances change, the service may need to be augmented, extended, refactored, or perhaps even replaced. Service-orientation design principles build native flexibility into service architectures so that, as software programs, services are resilient and adaptive to change and to being changed in response to real-world usage.

Separate the different aspects of a system that change at different rates.

What makes monolithic and silo-based systems inflexible is that change can have a significant impact on their existing usage. This is why it is often easier to create new silo-based applications rather than augment or extend existing ones.

The rationale behind the separation of concerns theory is that a larger problem can be more effectively solved when decomposed into a set of smaller problems or concerns. When applying service-orientation to the separation of concerns, we build corresponding units of solution logic that solve individual concerns, thereby allowing us to aggregate the units to solve the larger problem in addition to giving us the opportunity to aggregate them into different configurations in order to solve other problems.

Besides fostering service reusability, this approach introduces numerous layers of abstraction that help shield service-comprised systems from the impacts of change. This form of abstraction can exist at different levels. For example, if legacy resources encapsulated by one service need to be replaced, the impact of that change can be mitigated as long as the service is able to retain its original endpoint and functional behavior.

Another example is the separation of agnostic from non-agnostic logic. The former type of logic has high reuse potential if it is multi-purpose and less likely to change. Non-agnostic logic, on the other hand, typically represents the single-purpose parts of parent business process logic, which are often more volatile. Separating these respective logic types into different service layers further introduces abstraction that enables service reusability while shielding services, and any solutions that utilize them, from the impacts of change.

Reduce implicit dependencies and publish all external dependencies to increase robustness and reduce the impact of change.

This guiding principle embodies the purpose of the Service Loose Coupling (293) design principle. How a service architecture is internally structured and how services relate to programs that consume them (which can include other services) all comes down to dependencies that are formed on individually moving parts that are part of the service architecture.

Layers of abstraction help ease evolutionary change by localizing the impacts of the change to controlled regions. For example, within service architectures, service façades can be used to abstract parts of the implementation in order to minimize the reach of implementation dependencies.

On the other hand, published technical service contracts need to disclose the dependencies that service consumers must form in order to interact with services. As per the Service Abstraction (294) principle, the reduction of internal dependencies that can affect these technical contracts when change does occur minimizes the proliferation of the impact of those changes upon dependent service consumers.

At every level of abstraction, organize each service around a cohesive and manageable unit of functionality.

Each service requires a well-defined functional context that determines what logic does and does not belong within the service's functional boundary. Determining the scope and granularity of these functional service boundaries is one of the most critical responsibilities during the service delivery lifecycle.

Services with coarse functional granularity may be too inflexible to be effective, especially if they are expected to be reusable. On the other hand, overly fine-grained services may tax an infrastructure in that service compositions will need to consist of increased quantities of composition members.

Determining the right balance of functional scope and granularity requires a combination of business and technology expertise, and further requires an understanding of how services within a given boundary relate to each other.

Many of the guiding principles described in this manifesto help to make this determination in support of positioning each service as an IT asset that is capable of furthering an IT enterprise toward that target state whereby the strategic benefits of service-oriented computing are realized.

Ultimately, though, it is the attainment of real-world business value that dictates, from conception to delivery to repeated usage, the evolutionary path of any unit of service-oriented functionality.

About the Author

Thomas Erl

Thomas Erl is a top-selling IT author, founder of Arcitura Education, and series editor of the *Prentice Hall Service Technology Series from Thomas Erl*. With more than 300,000 copies in print worldwide, his books have become international bestsellers and have been formally endorsed by senior members of major IT organizations, such as IBM, Microsoft, Oracle, Intel, Accenture, IEEE, HL7, MITRE, SAP, CISCO, HP, and many others. As CEO of Arcitura Education Inc., Thomas has led the development of curricula for the internationally recognized Big Data Science Certified Professional (BDSCP), Cloud Certified Professional (CCP), and SOA Certified Professional (SOACP) accreditation programs, which have established a series of formal, vendor-neutral industry certifications obtained by thousands of IT professionals around the world. Thomas has toured more than 20 countries as a speaker and instructor. More than 100 articles and interviews by Thomas have been published in numerous publications, including *The Wall Street Journal* and *CIO Magazine*.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Prentice Hall Service Technology Series from Thomas Erl

THE WORLD'S TOP-SELLING SERVICE TECHNOLOGY TITLES

ABOUT THE SERIES

The Prentice Hall Service Technology 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 IT science and service technology application and innovation. Each title in this book series is authored in relation to other titles so as to establish a library of complementary knowledge. Although the series covers a broad spectrum of service technology-related topics, each title is authored in compliance with common language, vocabulary, and illustration conventions so as to enable readers to continually explore cross-topic research and education.

servicetechbooks.com/community

ABOUT THE SERIES EDITOR

Thomas Erl is a best-selling IT author, the series editor of the Prentice Hall Service Technology Series from Thomas Erl, and the editor of the Service Technology Magazine. As CEO of Arcitura Education Inc., Thomas has led the development of curricula for the internationally recognized Big Data Science Certified Professional (BDSCP), Cloud Certified Professional (CCP), and SOA Certified Professional (SOACP) accreditation programs, which have established a series of formal, vendor-neutral industry certifications. Thomas has toured over 20 countries as a speaker and instructor. Over 100 articles and interviews by Thomas have been published in numerous publications, including the Wall Street Journal and CIO Magazine.

informIT.com
THE TRUSTED TECHNOLOGY LEARNING SOURCE | PRENTICE HALL | ServiceTech PRESS

ALWAYS LEARNING

informit.com/erl

SOA with Java: Realizing Service-Orientation with Java Technologies
by T. Erl, S. Roy, P. Thomas, A. Tost
ISBN: 9780133859034
Hardcover, 592 pages

SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST
by R. Balasubramanian, B. Carlyle, T. Erl, C. Pautasso
ISBN: 0137012519
Hardcover, 577 pages

Cloud Computing Design Patterns
by T. Erl, R. Cope, A. Naserpour
ISBN: 9780133858563
Hardcover, 528 pages

Big Data Fundamentals: Concepts, Drivers & Techniques
by P. Buhler, T. Erl, W. Khattak
ISBN: 9780134291079
Paperback, 218 pages

Service-Oriented Architecture: Analysis & Design for Services and Microservices (Second Edition)
by T. Erl
ISBN: 0133858588
Paperback, ~ 300 pages

Web Service Contract Design & Versioning for SOA
by T. Erl, A. Karmarkar, P. Walmsley, H. Haas, U. Yalcinalp, C. Liu, D. Orchard, A. Tost, J. Pasley
ISBN: 013613517X
Hardcover, 826 pages

SOA Governance: Governing Shared Services On-Premise & in the Cloud
by S. Bennett, T. Erl, C. Gee, R. Laird, A. Manes, R. Schneider, L. Shuster, A. Tost, C. Venable
ISBN: 0138156751
Hardcover, 675 pages

SOA with .NET & Windows Azure: Realizing Service-Orientation with the Microsoft Platform
by D. Chou, J. deVadoss, T. Erl, N. Gandhi, H. Kommalapati, B. Loesgen, C. Schittko, H. Wilhelmsen, M. Williams
ISBN: 0131582313
Hardcover, 893 pages

Next Generation SOA: A Concise Introduction to Service Technology & Service-Orientation
by T. Erl, C. Gee, J. Kress, B. Maier, H. Normann, P. Raj, L. Shuster, B. Trops, C. Utschig-Utschig, P. Wik, T. Winterberg
ISBN: 9780133859041
Paperback, 208 pages

Cloud Computing: Concepts, Technology & Architecture
by T. Erl, Z. Mahmood, R. Puttini
ISBN: 9780133387520
Hardcover, 528 pages

Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services
by T. Erl
ISBN: 0131428985
Paperback, 534 pages

SOA Principles of Service Design
by T. Erl
ISBN: 0132344823
Hardcover, 573 pages

SOA Design Patterns
by T. Erl
ISBN: 0136135161
Hardcover, 865 pages

PEARSON VUE

The text books in this book series are official parts of Arcitura training and certification programs. All exams that correspond to associated courses are available at Pearson VUE testing centers and via Pearson VUE Online Proctoring. Visit www.pearsonvue.com/arcitura.

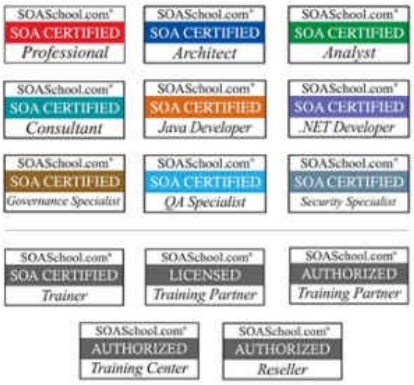
SOA & Cloud Computing Training & Certification

SOA Certified Professional (SOACP)

Content from this book and other series titles has been incorporated into the SOA Certified Professional (SOACP) program, an industry-recognized, vendor-neutral SOA certification curriculum developed by author Thomas Erl in cooperation with industry experts and academic communities and provided by SOASchool.com and training partners.

The SOA Certified Professional curriculum is comprised of a collection of 23 courses and labs that can be taken with or without formal testing and certification. Training can be delivered anywhere in the world by Certified Trainers. A comprehensive self-study program is available for remote, self-paced study, and exams can be taken world-wide via testing centers.

Dozens of public workshops are scheduled every quarter around the world by regional training partners.



All courses are reviewed and revised on a regular basis to stay in alignment with industry developments.

For more information, visit: **www.soaschool.com**

www.soaworkshops.com • www.soaselfstudy.com

Cloud Certified Professional (CCP)

The Certified Cloud Professional (CCP) program, provided by CloudSchool.com, establishes a series of vendor-neutral industry certifications dedicated to areas of specialization in the field of cloud computing. Also founded by author Thomas Erl, this program allows IT professionals to learn and become accredited in common and specialized topic areas within the field of cloud computing.

The Cloud Certified Professional curriculum is comprised of 21 courses and labs, each of which has a corresponding exam. Private and public training workshops can be provided throughout the world by certified Trainers. Self-study kits are further available for remote, self-paced study in support of instructor led workshops.

All courses are reviewed and revised on a regular basis to stay in alignment with industry developments.

For more information, visit: **www.cloudschool.com**

www.cloudworkshops.com • www.cloudselfstudy.com

Arcitura™
the IT education company

PEARSON VUE
All Arcitura exams are available at Pearson VUE testing centers and via Pearson VUE Online Proctoring

Code Snippets

```
<xml:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace=
    "http://www.example.org/tls/employee/schema/accounting/">
  <xml:element name="EmployeeHoursRequestType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ID" type="xml:integer"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
  <xml:element name="EmployeeHoursResponseType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ID" type="xml:integer"/>
        <xml:element name="WeeklyHoursLimit"
          type="xml:short"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
</xml:schema>
```

```
<xml:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace=
    "http://www.example.org/tls/employee/schema/hr/">
  <xml:element name="EmployeeUpdateHistoryRequestType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ID" type="xml:integer"/>
        <xml:element name="Comment" type="xml:string"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
  <xml:element name="EmployeeUpdateHistoryResponseType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ResponseCode"
          type="xml:byte"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
</xml:schema>
```

```
<types>
  <xml:schema targetNamespace=
    "http://www.example.org/tls/employee/schema/">
    <xml:import namespace=
      "http://www.example.org/tls/employee/schema/accounting/"
      schemaLocation="Employee.xsd"/>
    <xml:import namespace=
      "http://www.example.org/tls/employee/schema/hr/"
      schemaLocation="EmployeeHistory.xsd"/>
  </xml:schema>
</types>
```

```
<message name="getEmployeeWeeklyHoursRequestMessage">
  <part name="RequestParameter"
    element="act:EmployeeHoursRequestType"/>
</message>
<message name="getEmployeeWeeklyHoursResponseMessage">
  <part name="ResponseParameter"
    element="act:EmployeeHoursResponseType"/>
</message>
<message name="updateEmployeeHistoryRequestMessage">
  <part name="RequestParameter"
    element="hr:EmployeeUpdateHistoryRequestType"/>
</message>
<message name="updateEmployeeHistoryResponseMessage">
  <part name="ResponseParameter"
    element="hr:EmployeeUpdateHistoryResponseType"/>
</message>
<portType name="EmployeeInterface">
  <operation name="GetEmployeeWeeklyHoursLimit">
    <input message=
      "tns:getEmployeeWeeklyHoursRequestMessage"/>
    <output message=
      "tns:getEmployeeWeeklyHoursResponseMessage"/>
  </operation>
  <operation name="UpdateEmployeeHistory">
    <input message=
      "tns:updateEmployeeHistoryRequestMessage"/>
    <output message=
      "tns:updateEmployeeHistoryResponseMessage"/>
  </operation>
</portType>
```

```
<portType name="EmployeeInterface">
  <documentation>
    GetEmployeeWeeklyHoursLimit uses the Employee
    ID value to retrieve the WeeklyHoursLimit value.
    UpdateEmployeeHistory uses the Employee ID value
    to update the Comment value of the EmployeeHistory.
  </documentation>
  <operation name="GetEmployeeWeeklyHoursLimit">
    <input message=
      "tns:getEmployeeWeeklyHoursRequestMessage"/>
    <output message=
      "tns:getEmployeeWeeklyHoursResponseMessage"/>
  </operation>
  <operation name="UpdateEmployeeHistory">
    <input message=
      "tns:updateEmployeeHistoryRequestMessage"/>
    <output message=
      "tns:updateEmployeeHistoryResponseMessage"/>
  </operation>
</portType>
```

```
<operation name="GetWeeklyHoursLimit">
  <input message="tns:getWeeklyHoursRequestMessage"/>
  <output message="tns:getWeeklyHoursResponseMessage"/>
</operation>
<operation name="UpdateHistory">
  <input message="tns:updateHistoryRequestMessage"/>
  <output message="tns:updateHistoryResponseMessage"/>
</operation>
```

```
<definitions name="Employee"
  targetNamespace="http://www.example.org/tls/employee/wsdl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:act=
    "http://www.example.org/tls/employee/schema/accounting/"
  xmlns:hr="http://www.example.org/tls/employee/schema/hr/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.example.org/tls/employee/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <xml:schema targetNamespace=
      "http://www.example.org/tls/employee/schema/">
      <xml:import namespace=
        "http://www.example.org/tls/employee/schema/
          accounting/"
        schemaLocation="Employee.xsd"/>
      <xml:import namespace=
        "http://www.example.org/tls/employee/schema/hr/"
        schemaLocation="EmployeeHistory.xsd"/>
    </xml:schema>
  </types>
  <message name="getWeeklyHoursRequestMessage">
    <part name="RequestParameter"
      element="act:EmployeeHoursRequestType"/>
  </message>
  <message name="getWeeklyHoursResponseMessage">
    <part name="ResponseParameter"
      element="act:EmployeeHoursResponseType"/>
```

```
  </message>
  <message name="updateHistoryRequestMessage">
    <part name="RequestParameter"
      element="hr:EmployeeUpdateHistoryRequestType"/>
  </message>
  <message name="updateHistoryResponseMessage">
    <part name="ResponseParameter"
      element="hr:EmployeeUpdateHistoryResponseType"/>
  </message>
  <portType name="EmployeeInterface">
    <documentation>
      GetWeeklyHoursLimit uses the Employee ID value
      to retrieve the WeeklyHoursLimit value.
      UpdateHistory uses the Employee ID value to
      update the Comment value of the EmployeeHistory.
    </documentation>
    <operation name="GetWeeklyHoursLimit">
      <input message=
        "tns:getWeeklyHoursRequestMessage"/>
      <output message=
        "tns:getWeeklyHoursResponseMessage"/>
    </operation>
    <operation name="UpdateHistory">
      <input message=
        "tns:updateHistoryRequestMessage"/>
      <output message=
        "tns:updateHistoryResponseMessage"/>
    </operation>
  </portType>
  ...
</definitions>
```

```
<import namespace="http://.../common/wsdl/"
  location="http://.../common/wsdl/bindings.wsdl"/>
```

http://schemas.xmlsoap.org/wsdl/
http://schemas.xmlsoap.org/wsdl/soap/
http://www.w3.org/2001/XMLSchema/
http://schemas.xmlsoap.org/wsdl/http/
http://schemas.xmlsoap.org/wsdl/mime/
http://schemas.xmlsoap.org/soap/envelope/

```
<definitions name="Employee"
  targetNamespace="http://www.xmltc.com/tls/employee/wsdl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:act=
    "http://www.xmltc.com/tls/employee/schema/accounting/"
  xmlns:hr="http://www.xmltc.com/tls/employee/schema/hr/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.xmltc.com/tls/employee/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ...
</definitions>
```

```
<binding name="EmployeeBinding"
  type="tns:EmployeeInterface">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetWeeklyHoursLimit">
    <soap:operation
      soapAction="http://www.xmltc.com/soapaction"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="UpdateHistory">
    <soap:operation
      soapAction="http://www.xmltc.com/soapaction"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
```

application/vnd.edu.mua.student-award-conferral-application+xhtml+xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" >
  <head>
    <title>Student Award Conferral Application</title>
  </head>
  <body>
    <p>Student:
      <a rel="student"
        href="http://student.mua.edu/student/555333">
        John Smith (Student Number 555333)
      </a>
    </p>
    <p>Award:
      <a rel="award"
        href="http://award.mua.edu/award/BS/CompSci">
        Bachelor of Science with Computer Science Major
      </a>
    </p>
    <p>Event:
      <a rel="event"
        href="http://event.mua.edu/achievement">
        Outstanding Achievement
      </a>
    </p>
  </body>
</html>
```

application/vnd.*organization.type+supertype*

```
Media type = application/vnd.com.actioncon.po+xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://example.org/schema/po"
  xmlns="http://example.org/schema/po">
  <xsd:element name="LineItemList" type="LineItemListType"/>
  <xsd:complexType name="LineItemListType">
    <xsd:element name="LineItem" type="LineItemType"
      minOccurs="0"/>
  </xsd:complexType>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:anyURI"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```
<xsd:element name="LineItem" type="LineItemType"/>
<xsd:complexType name="LineItemType">
  <xsd:sequence>
    <xsd:element name="productID" type="xsd:string"/>
    <xsd:element name="productName" type="xsd:string"/>
    <xsd:any minOccurs="0" maxOccurs="unbounded"
      namespace="##any" processContents="lax"/>
  </xsd:sequence>
  <xsd:anyAttribute namespace="##any"/>
</xsd:complexType>
```

```
<definitions name="Purchase Order" targetNamespace=
  "http://actioncon.com/contract/po"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://actioncon.com/contract/po"
  xmlns:po="http://actioncon.com/schema/po">
  ...
  <portType name="ptPurchaseOrder">
    <operation name="opSubmitOrder">
      <input message="tns:msgSubmitOrderRequest"/>
      <output message="tns:msgSubmitOrderResponse"/>
    </operation>
    <operation name="opCheckOrderStatus">
      <input message="tns:msgCheckOrderRequest"/>
      <output message="tns:msgCheckOrderResponse"/>
    </operation>
    <operation name="opChangeOrder">
      <input message="tns:msgChangeOrderRequest"/>
      <output message="tns:msgChangeOrderResponse"/>
    </operation>
    <operation name="opCancelOrder">
      <input message="tns:msgCancelOrderRequest"/>
      <output message="tns:msgCancelOrderResponse"/>
    </operation>
    <operation name="opGetOrder">
      <input message="tns:msgGetOrderRequest"/>
      <output message="tns:msgGetOrderResponse"/>
    </operation>
  </portType>
</definitions>
```

```
Service: po.actioncon.com
Capabilities:
POST /orders
      In = application/vnd.com.actioncon.po+xml
GET /orders/{order-id}/status
      Out = text/plain
PUT /orders/{order-id}
      In = application/vnd.com/actioncon.po+xml
DELETE /orders/{order-id}
GET /orders/{order-id}
      Out = application/vnd.com.actioncon.po+xml
```

Legal methods for actioncon.com service inventory:
* GET
* PUT
* DELETE
* POST
* **SUBSCRIBE** (consumers must fall back to periodic GET if service reports "not implemented")

```
Media type = application/vnd.com.actioncon.po+xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:any namespace="##any" processContents="lax"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:anyAttribute namespace="##any"/>
  </xsd:complexType>
</xsd:schema>
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"
        minOccurs="0"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"
        minOccurs="3"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="3"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```
<xsd:schema version="2.0" ...>
```

```
<LineItem xmlns="http://actioncon.com/schema/po/v2">
```

application/vnd.com.actioncon.po.v2+xml

```
<LineItem xmlns="http://actioncon.com/schema/po/2010/09">
```