SOA Design Patterns

000_0136135161_FM.qxd 11/24/08 4:43 PM Page ix

SOA Design Patterns

Thomas Erl

(with additional contributors)

PRENTICE HALL

PRENTICE HALL

UPPER SADDLE RIVER, NJ • BOSTON • INDIANAPOLIS • SAN FRANCISCO

NEW YORK • TORONTO • MONTREAL • LONDON • MUNICH • PARIS • MADRID

CAPETOWN • SYDNEY • TOKYO • SINGAPORE • MEXICO CITY

Library of Congress Cataloging-in-Publication Data:

Erl, Thomas.

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

p. cm.

ISBN 0-13-613516-1 (hardback : alk. paper) 1. Web services.
2. Computer architecture. 3. Software patterns. 4. System design. I. Title.

TK5105.88813.E735 2008

006.7—dc22

2008040488

ISBN-13: 978-0-13-613516-6
ISBN-10: 0-13-613516-1
Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing December 2008

The following patterns: Exception Shielding, Threat Screening, Trusted Subsystem, Service Perimeter Guard, Data Confidentiality, Data Origin Authentication, Direct Authentication, Brokered Authentication are courtesy of the Microsoft Patterns & Practices team. For more information please visit http://msdn.microsoft.com/practices.
These patterns were originally developed by Jason Hogg, Frederick Chong, Dwayne Taylor, Lonnie Wall, Paul Slater, Tom Hollander, Wojtek Kozaczynski, Don Smith, Larry Brader, Sajjas Nasir Imran, Pablo Cibraro, Nelly Delgado and Ward Cunningham

Editor-in-Chief
Mark L. Taub

Managing Editor
Kristy Hart

Copy Editor
Language Logistics

Indexer
Cheryl Lenser

Proofreader
Williams Woods
Publishing

Composition
Jake McFarland
Bumpy Design

Graphics
Zuzana Cappova
Tami Young
Spencer Fruhling

Photos
Thomas Erl

Cover Design
Thomas Erl

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

- Thomas Erl

Contents

000_0136135161_FM.qxd 11/24/08 4:43 PM Page xviii

000_0136135161_FM.qxd 11/24/08 4:43 PM Page xxxii

Foreword

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

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

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

Thomas' work is therefore the right book at the right time. He really groks the nature of SOA systems: There are many hard design decisions to be made, ranging from data-orientation to the problems of legacy integration and even security. Thomas offers wise counsel on each of these issues and many more, all in the language of design patterns. There are many things I like about this work. It's comprehensive. It's written in a very accessible pattern language. It offers patterns that play well with one another. Finally, Thomas covers not just the technical details, but also sets these patterns in the context of economic and other considerations.

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

—*Grady Booch, IBM Fellow*
September, 2008

Acknowledgments

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

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

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

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

Mohamad Afshar, Oracle

Sanjay Agara, Wipro

Stephen Bennett, Oracle

Steve Birkel, Intel

Brandon Bohling, Intel

Grady Booch, IBM

Bryan Brew, Booz Allen Hamilton

Victor Brown, CMGC

Frank Buschmann, Siemens

Enrique G. Castro-Leon, Intel

Peter Chang, Lawrence Technical University

Jason "AJ" Comfort, Booz Allen Hamilton

John Crupi, JackBe

Veronica Gacitua Decar, Dublin City University

Ed Dodds, Conmergence

Kevin P. Davis, PhD

Dominic Duggan, Stevens Institute of Technology

Baptist Eggen, Dutch Department of Defense

Steve Elston, Microsoft

Dale Ferrario, Sun Microsystems

Martin Fowler, ThoughtWorks

Pierre Fricke, Red Hat

Chuck Georgo, Public Safety and National Security

Larry Gloss, Information Manufacturing

Al Gough, CACI International Inc.

Daniel Gross, University of Toronto

Robert John Hathaway III, SOA Object Systems

William M. Hegarty, ThoughtWorks

Gregor Hohpe, Google

Ralph Johnson, UIUC

James Kinneavy, University of California

Robert Laird, IBM

Doug Lea, Oswego State University of New York

Canyang Kevin Liu, SAP

Terry Lottes, Northrop Grumman Mission Systems

Chris Madrid, Microsoft

Anne Thomas Manes, Burton Group

Dragos Manolescu, Microsoft

Steven Martin, Microsoft

Joe McKendrick

J.D. Meier, Microsoft

David Michalowicz, MITRE Corporation

Per Vonge Nielsen, Microsoft

Wendell Ocasio, DoD Military Health Systems, Agilex Technologies

Philipp Offermann, University of Berlin

Dmitri Ossipov, Microsoft

Prasen Palvakar, Oracle

Parviz Peiravi, Intel

Nishit Rao, Oracle

Ian Robinson, ThoughtWorks

Richard Van Schelven, Ericsson

Shakti Sharma, Sysco Corp

Don Smith, Microsoft

Michael Sor, Booz Allen Hamilton

John Sparks, Western Southern Life

Sona Srinivasan, CISCO

Linda Terlouw, Ordina

Phil Thomas, IBM

Steve Vinoski, IEEE

Herbjörn Wilhelmsen, Objectware

Peter B. Woodhull, Modus21

Bobby Woolf, IBM

Farzin Yashar, IBM

Markus Zirn, Oracle

Olaf Zimmermann, IBM

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

Contributors

In alphabetical order by last name:

Larry Brader

David Chappell, Oracle

Frederick Chong

Pablo Cibraro, Lagash Systems SA

Ward Cunningham

Nelly Delgado, Microsoft

Florent Georges

Charles Stacy Harris, Microsoft

Kelvin Henney, Curbralan

Jason Hogg, Microsoft

Tom Hollander

Anish Karmarkar, Oracle

Sajjas Nasir Imran, Infosys

Berthold Maier, Oracle

Hajo Normann, EDS

Wojtek Kozaczynski

Mark Little, Red Hat

Brian Lokhorst, Dutch Tax Office

Brian Loesgen, Neudesic

Matt Long, Microsoft

David Orchard, Oracle

Thomas Rischbeck, IPT

Chris Riley, SOA Systems

Satadru Roy, Sun Microsystems

Arnaud Simon, Red Hat

Paul Slater, Wadeware

Don Smith

Sharon Smith, Microsoft

Dwayne Taylor

Tina Tech

Bernd Trops, SOPERA GmbH

Clemens Utschig-Utschig, Oracle

Lonnie Wall, RDA Corporation

Torsten Winterberg, Oracle

Dennis Wisnosky, U.S. Department of Defense

Chapter 16

Service Governance Patterns

Compatible Change

Version Identification

Termination Notification

Service Refactoring

Service Refactoring

Service Decomposition

Proxy Capability

Decomposed Capability

Distributed Capability

NOTE

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

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

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

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

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

Compatible Change

By David Orchard, Chris Riley

How can a service contract be modified without impacting consumers?

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

Table 16.1
Profile summary for the Compatible Change pattern.

Problem

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

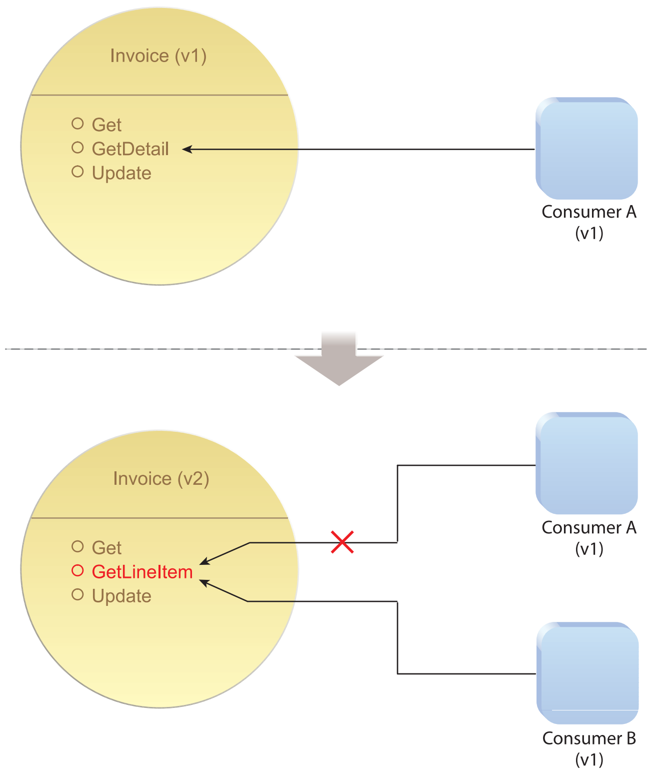


Figure 16.1
The name of a service capability is modified after version 1 of a service contract is already in use. As a result, version 2 of the contract is incompatible with Consumer A.

Solution

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

Application

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

019_0136135161_16.qxd 11/24/08 4:52 PM Page 467

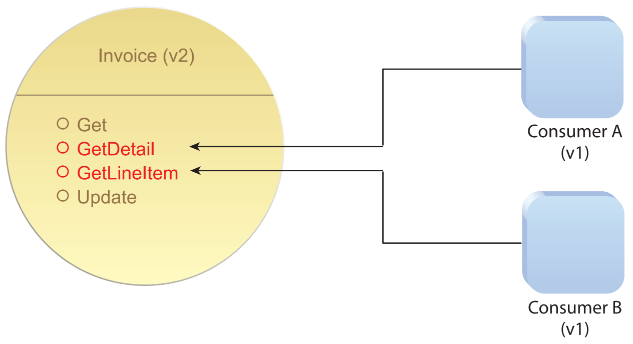


Figure 16.2
The existing capability is not renamed. Instead, a new capability with a new name is added alongside the original capability, thereby preserving compatibility with both Consumers A and B.

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

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

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

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

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

NOTE

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

Impacts

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

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

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

Relationships

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

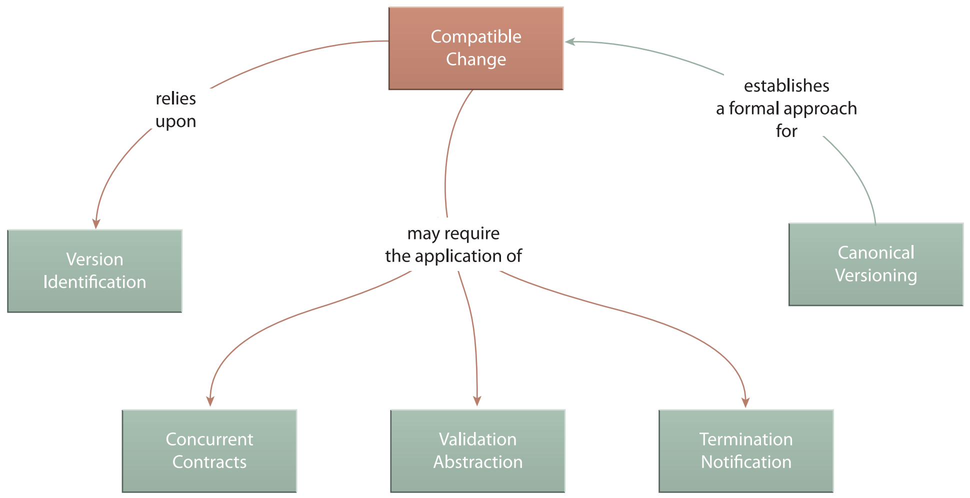


Figure 16.3
Compatible Change relates to several other service governance patterns but also may depend on some contract design patterns.

CASE STUDY EXAMPLE

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

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

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

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

Example 16.1

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

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

Version Identification

By David Orchard, Chris Riley

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

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

Table 16.2
Profile summary for the Version Identification pattern.

Problem

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

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

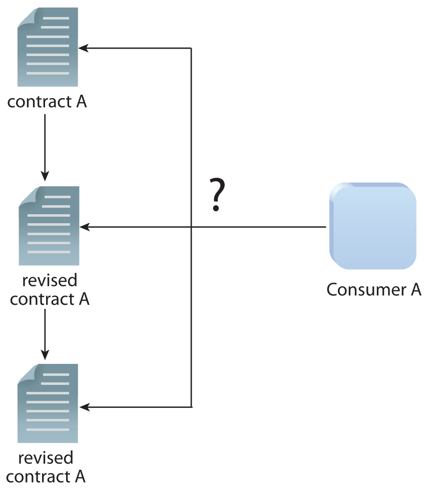


Figure 16.4
As a service contract is required to change, a service consumer is left in the dark as to whether it is still compatible.

Solution

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

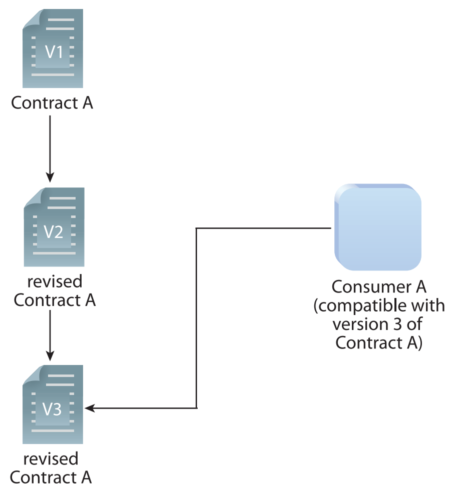


Figure 16.5

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

Application

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

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

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

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

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

NOTE

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

Impacts

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

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

Relationships

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

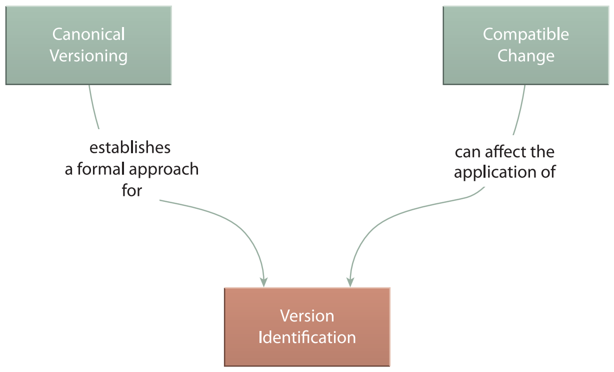

Canonical Versioning
establishes a formal approach for
Compatible Change
can affect the application of
Version Identification

Figure 16.6

Version Identification relates primarily to other contract versioning patterns.

NOTE

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

CASE STUDY EXAMPLE

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

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

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

Example 16.2
The WSDL definition from Example 16.1 is annotated with a version number.

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

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

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

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

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

Example 16.3

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

Termination Notification

By David Orchard, Chris Riley

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

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

Table 16.3
Profile summary for the Termination Notification pattern.

Problem

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

Examples include:

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

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

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

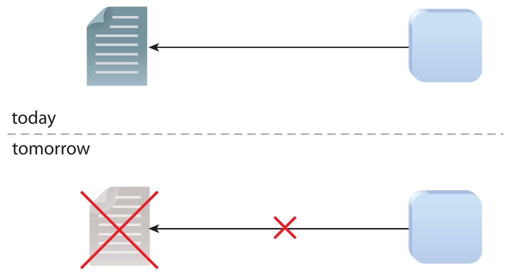


Figure 16.7
The consumer program (right) invokes a service via its contract as usual today, but when the contract is terminated on the next day, the attempted invocation fails.

Solution

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

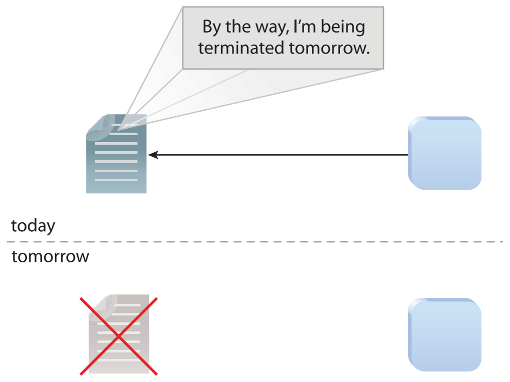


Figure 16.8
The service contract includes a standardized statement that communicates when it is scheduled for termination. As a result, the consumer does not attempt to invoke it after the contract has been terminated.

Application

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

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

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

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

Impacts

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

Relationships

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

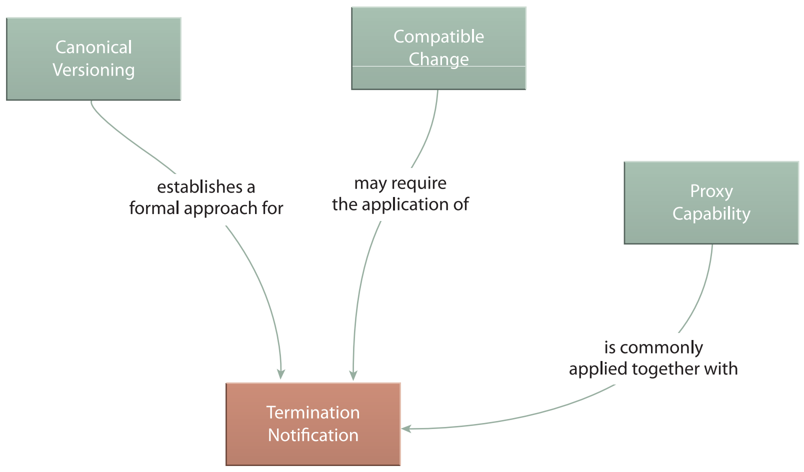

Canonical Versioning
Compatible Change
establishes a formal approach for
may require the application of
Proxy Capability
is commonly applied together with
Termination Notification

Figure 16.9

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

NOTE

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

CASE STUDY EXAMPLE

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

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

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

Step 3 is implemented as follows:

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

019_0136135161_16.qxd 11/24/08 4:52 PM Page 483

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

Example 16.4

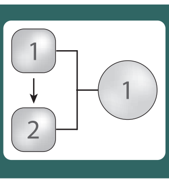
The `operation` element within the Officer WSDL definition's `binding` construct is modified to include a custom ignorable WS-Policy assertion that expresses the scheduled termination date of the UpdateLog operation.

NOTE

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

Service Refactoring

How can a service be evolved without impacting existing consumers?



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

Table 16.4
Profile summary for the Service Refactoring pattern.

Problem

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

019_0136135161_16.qxd 11/24/08 4:52 PM Page 485

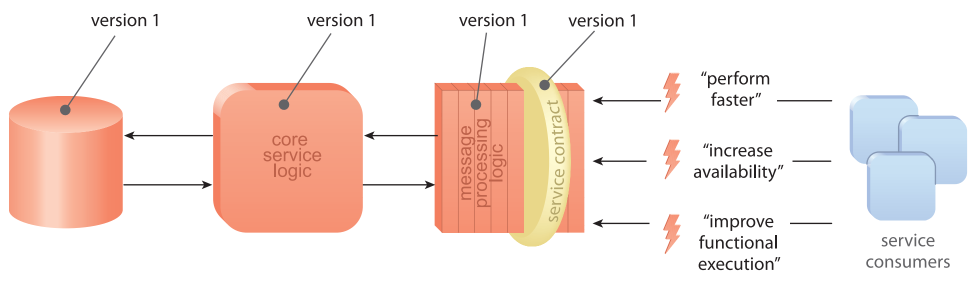


Figure 16.10
Consumers of an existing service demand new requirements for which the service was not originally designed. The red symbols indicate the different parts of this service architecture that could be independently versioned.

Solution

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

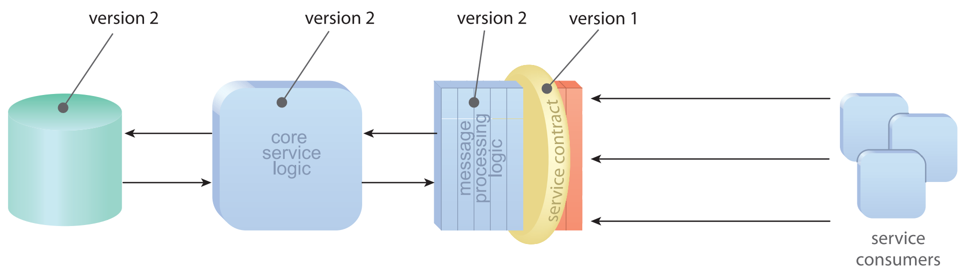


Figure 16.11
All parts of a service architecture abstracted by its contract can potentially be refactored without compromising existing consumer relationships. The service contract and the remaining, externally facing message processing agents (red) are not affected by the refactoring effort.

Application

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

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

NOTE

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

Impacts

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

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

Relationships

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

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

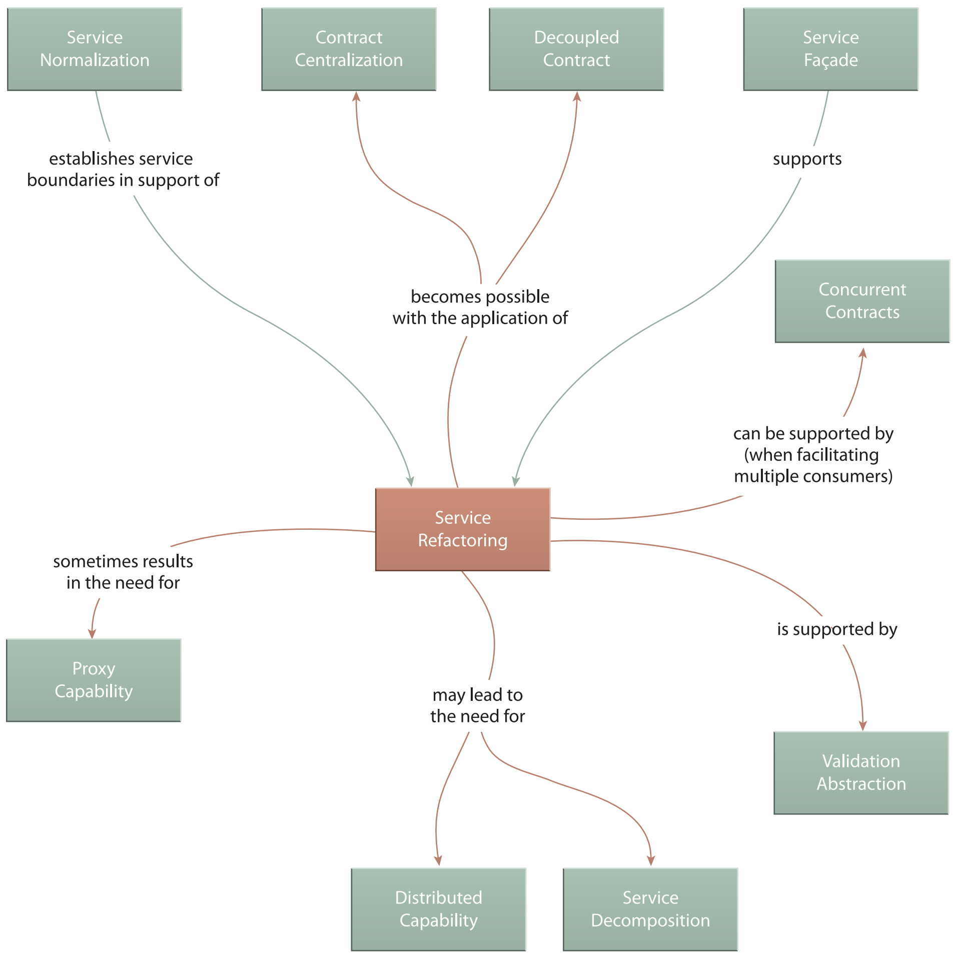


Figure 16.12
Service Refactoring relies on several key contract-related patterns to ensure that refactoring-related changes do not disrupt existing service consumers.

CASE STUDY EXAMPLE

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

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

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

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

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

Service Decomposition

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

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

Table 16.5
Profile summary for the Service Decomposition pattern.

Problem

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

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

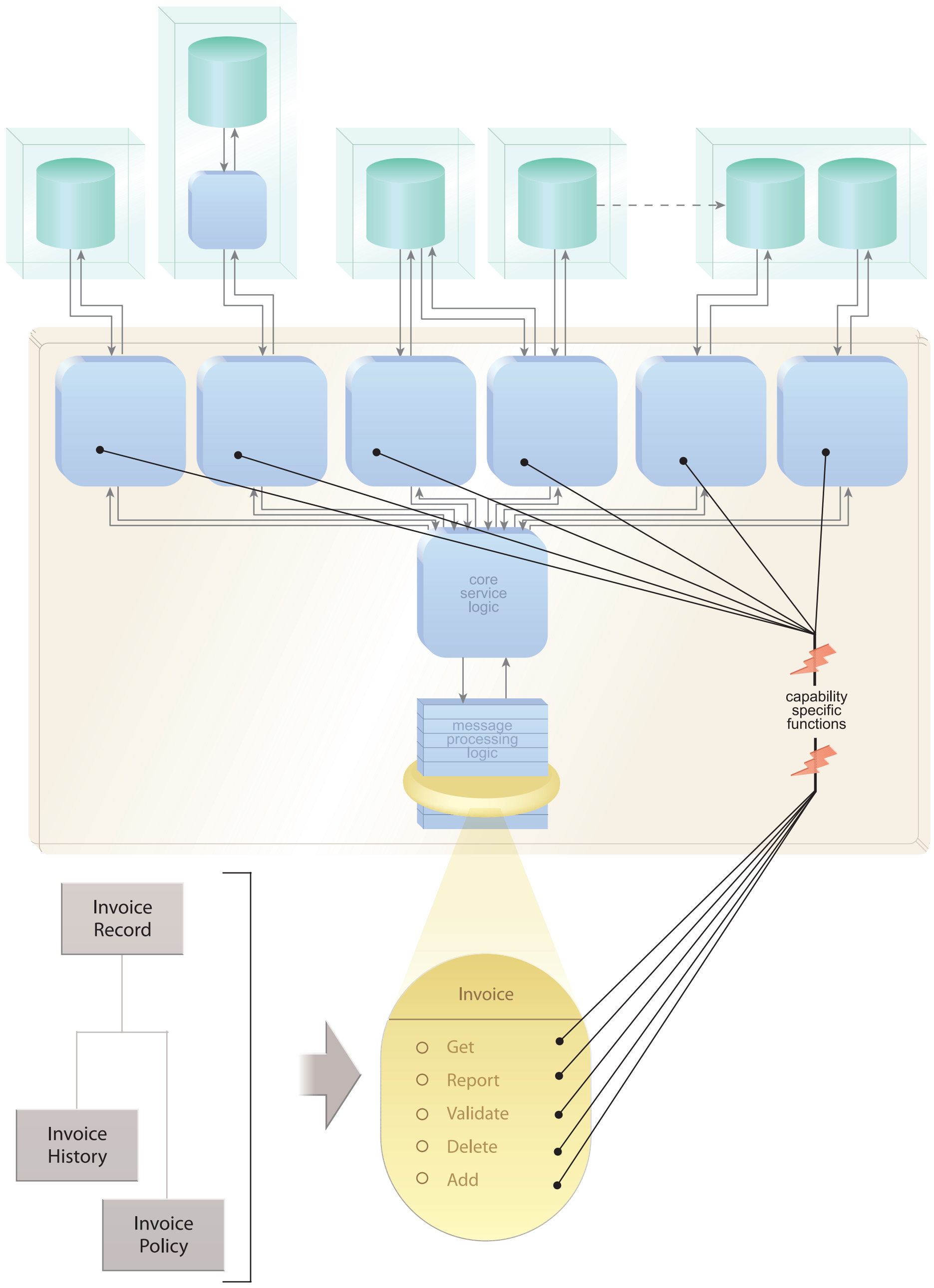


Figure 16.13

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

NOTE

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

Solution

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

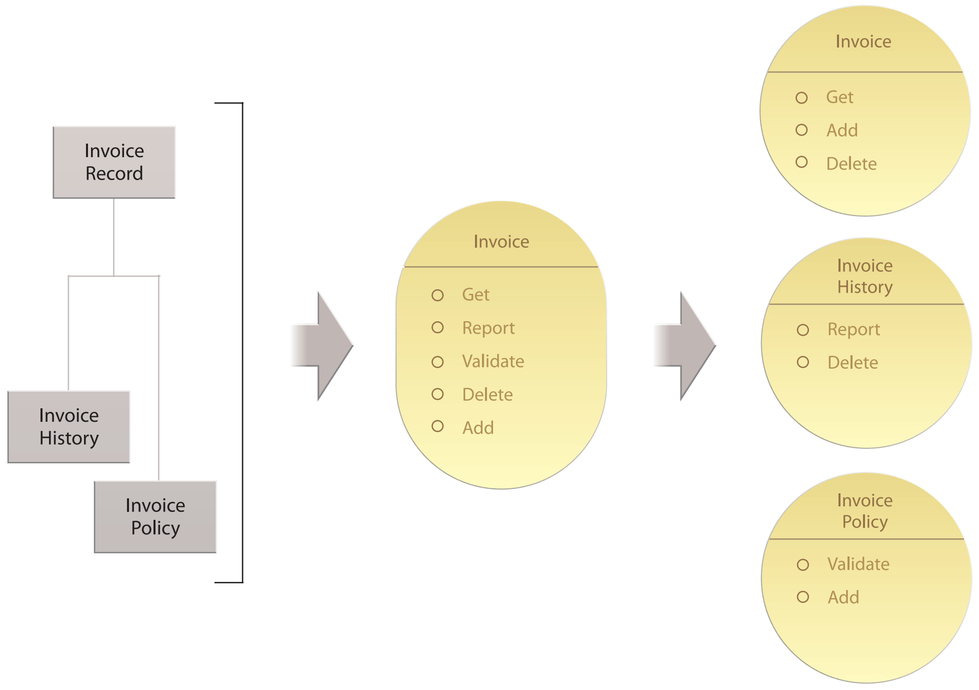


Figure 16.14

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

Application

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

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

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

NOTE

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

Impacts

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

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

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

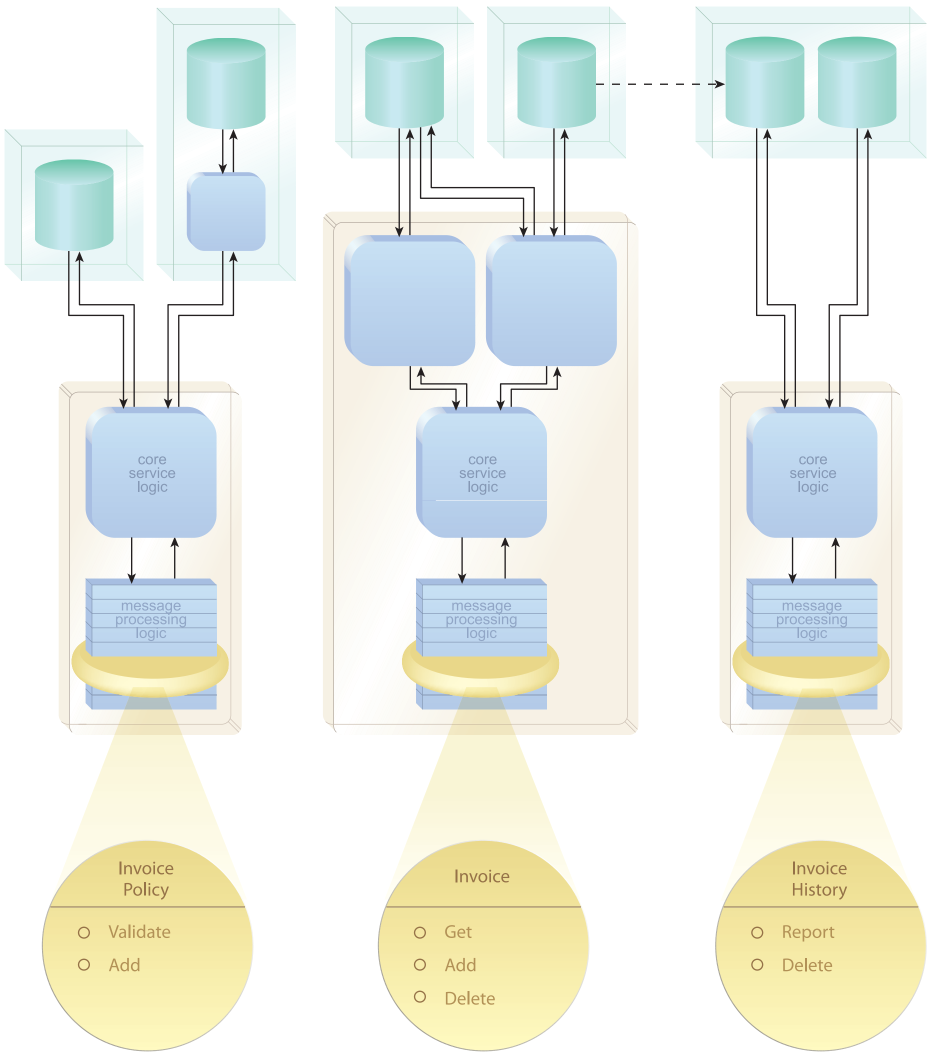


Figure 16.15
The new, fine-grained services each provide fewer capabilities and therefore also impose smaller program sizes.

Relationships

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

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

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

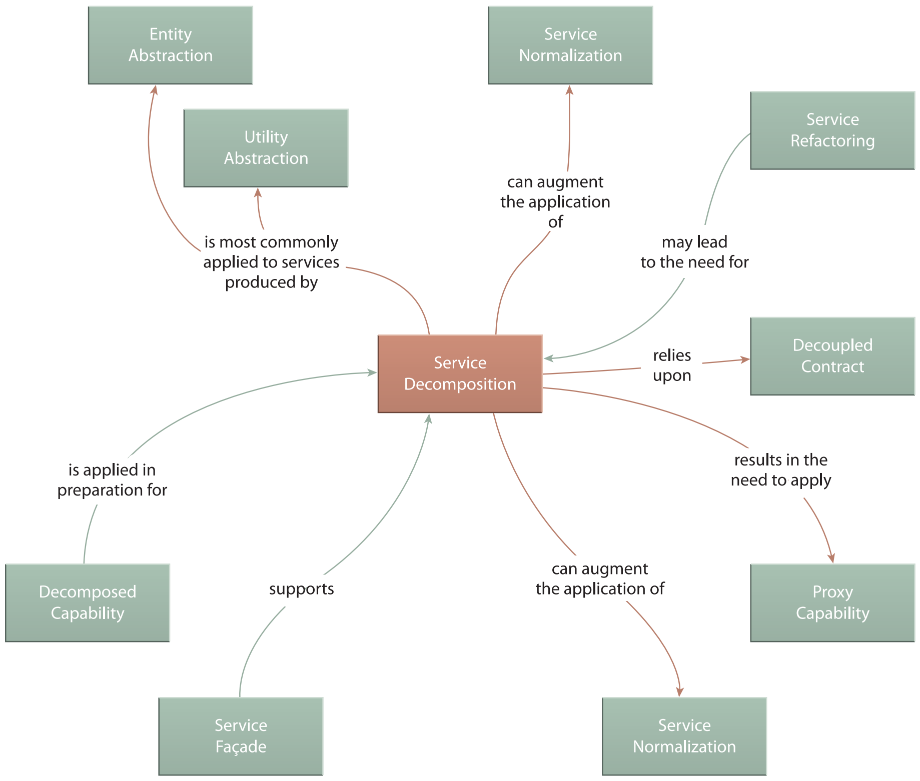


Figure 16.16

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

CASE STUDY EXAMPLE

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

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

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

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

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

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

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

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

NOTE

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

Proxy Capability

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

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

Table 16.6
Profile summary for the Proxy Capability pattern.

Problem

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

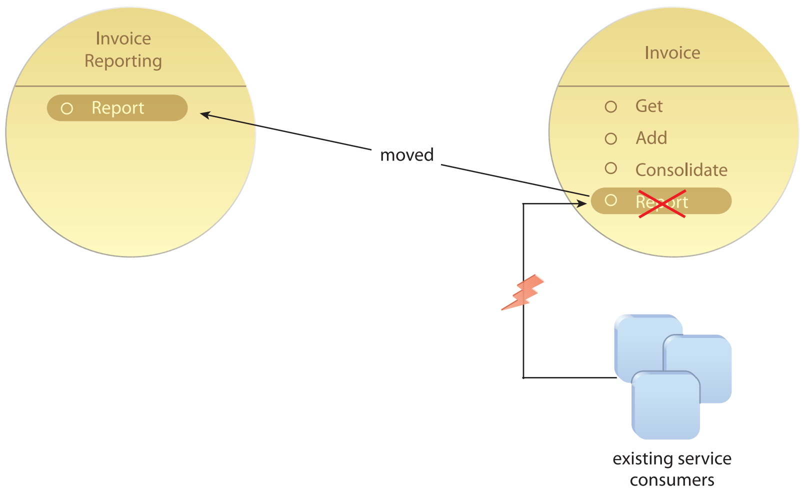


Figure 16.17
Moving a service capability that is part of an established service contract will predictably impact existing service consumers.

Solution

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

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

Application

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

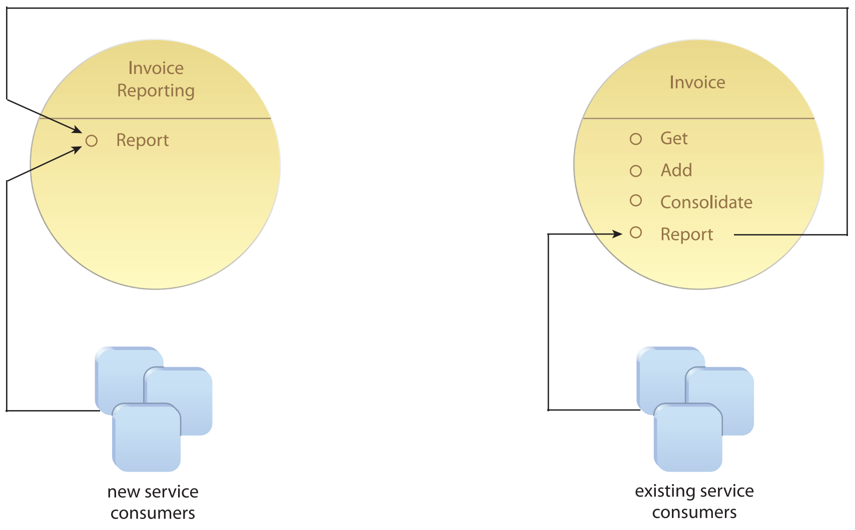


Figure 16.18

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

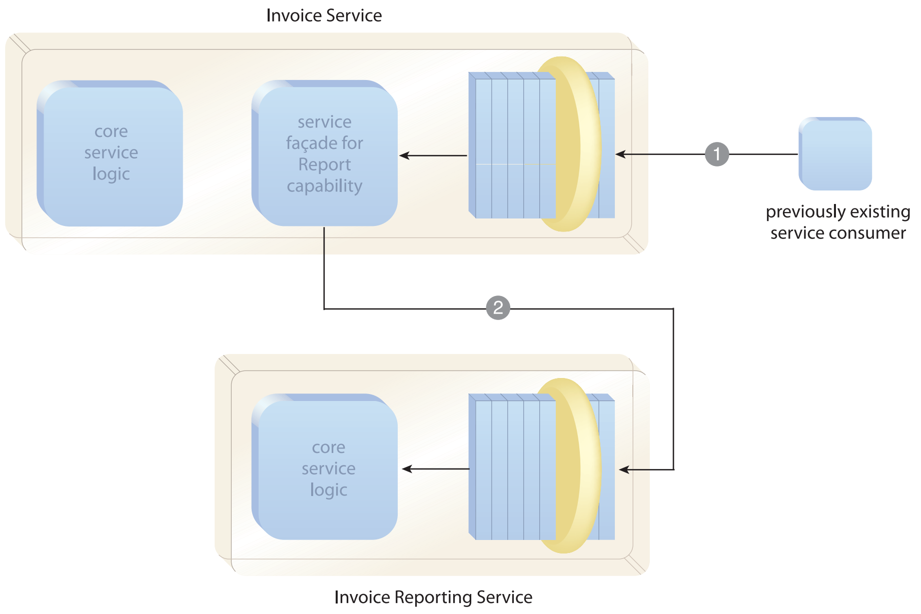

Invoice Service
core service logic
service façade for Report capability
1
previously existing service consumer
2
core service logic
Invoice Reporting Service

Figure 16.19

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

019_0136135161_16.qxd 11/24/08 4:52 PM Page 500

NOTE

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

Impacts

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

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

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

Relationships

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

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

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

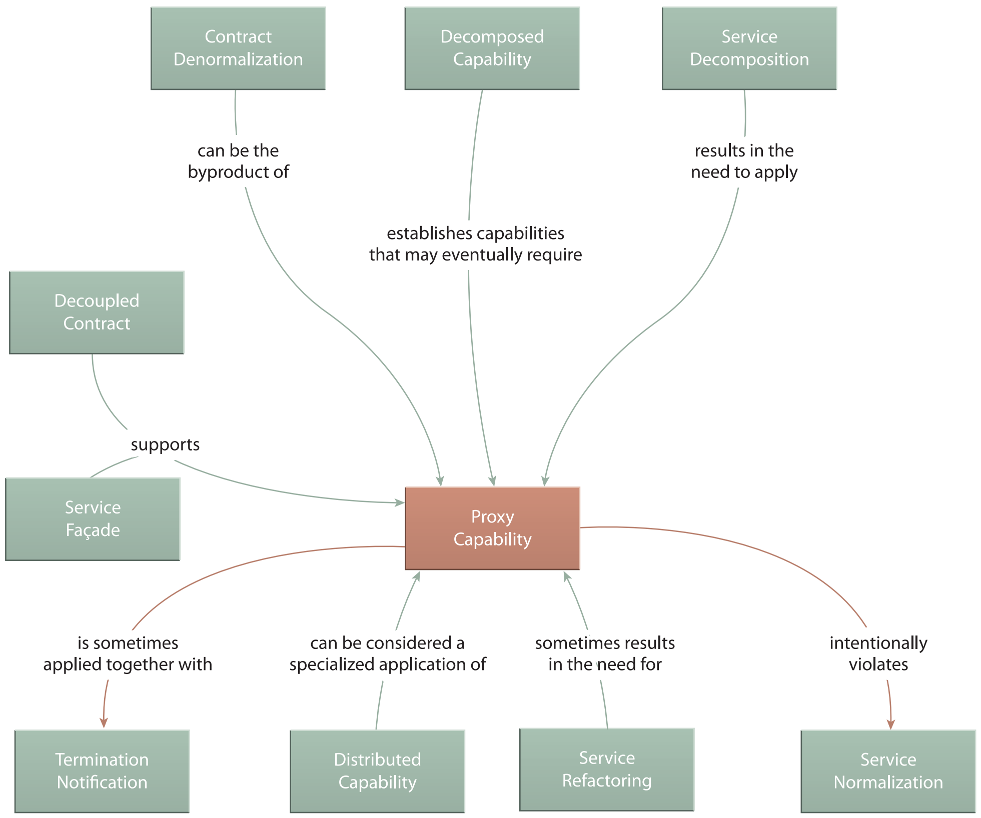


Figure 16.20
Proxy Capability alters the structure of a service in support of the creation of a new service and therefore touches several patterns related to service logic structure and the service decomposition process.

CASE STUDY EXAMPLE

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

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

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

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

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

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

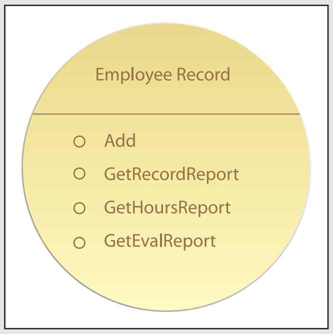


Figure 16.21
After some analysis, the new Employee Record service candidate is modeled with four capability candidates.

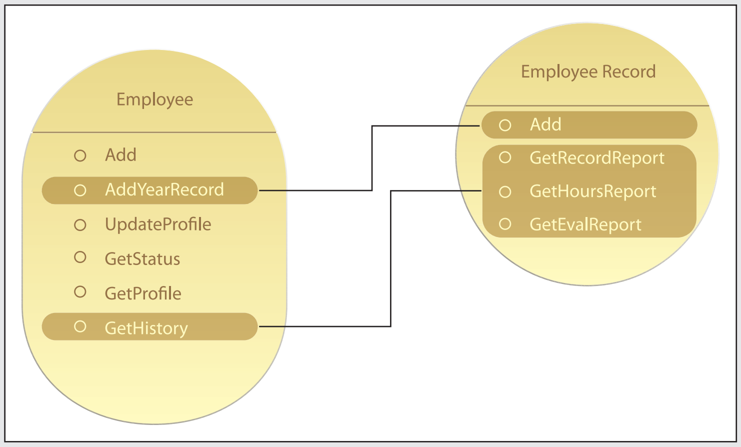


Figure 16.22
The Employee service's AddRecord and GetHistory capabilities are positioned as proxies for the Employee Record's Add, GetRecordReport, GetHoursReport, and GetEvalReport capabilities.

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

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

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

Decomposed Capability

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

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

Table 16.7
Profile summary for the Decomposed Capability pattern.

Problem

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

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

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

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

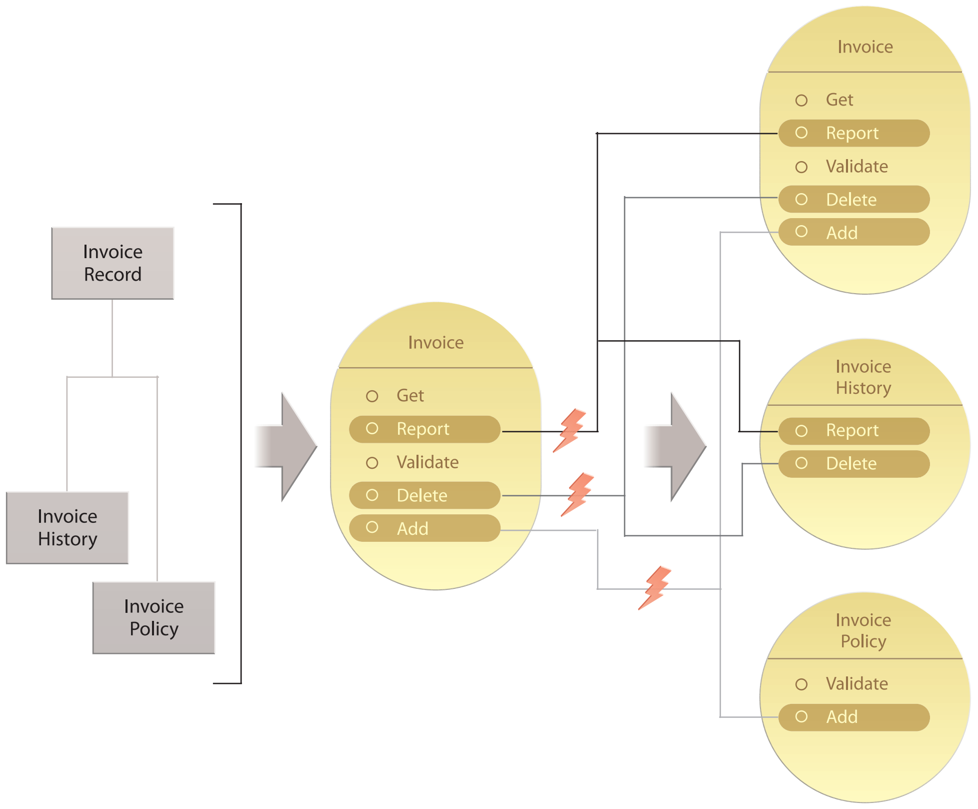


Figure 16.23
An Invoice entity service (middle) derived from a group of Invoice-related business entities (left) exposes coarse-grained capabilities that are difficult to decompose when service decomposition requirements present themselves. Each of the affected Invoice service capabilities needs to be split up in order to accommodate the new services (right).

Solution

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

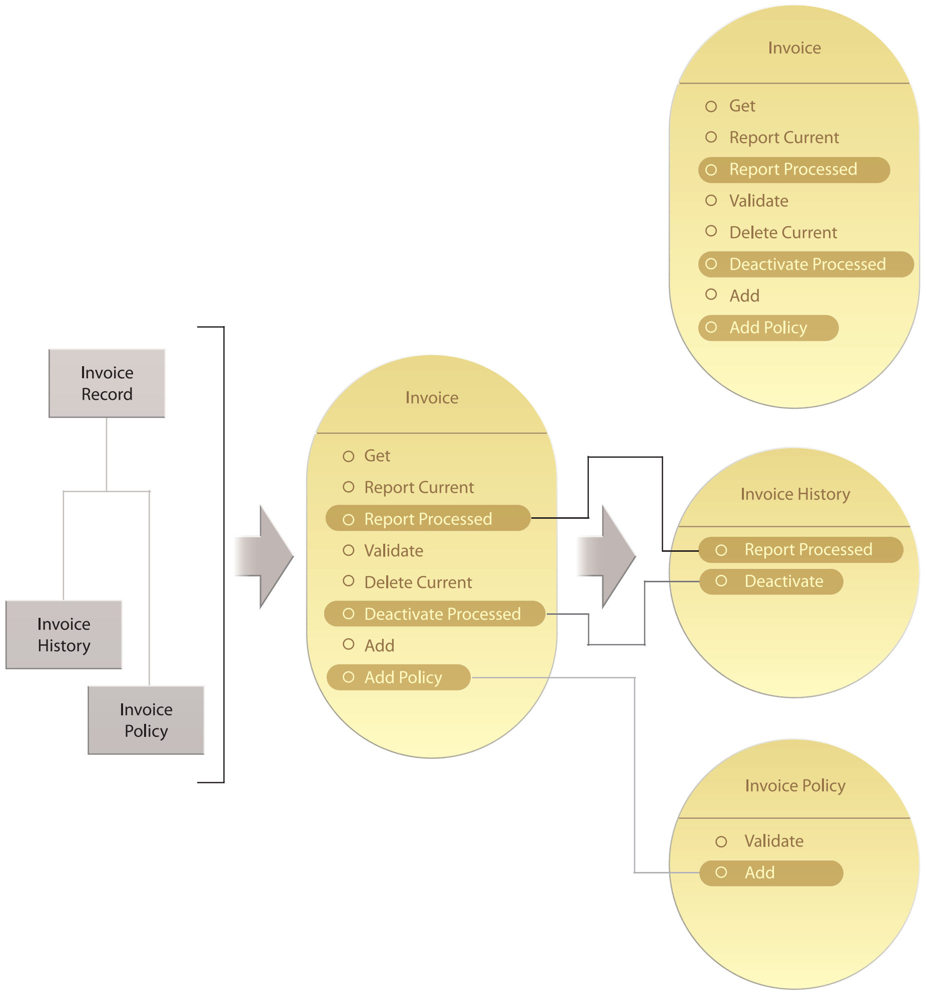


Figure 16.24

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

Application

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

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

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

NOTE

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

Impacts

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

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

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

Relationships

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

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

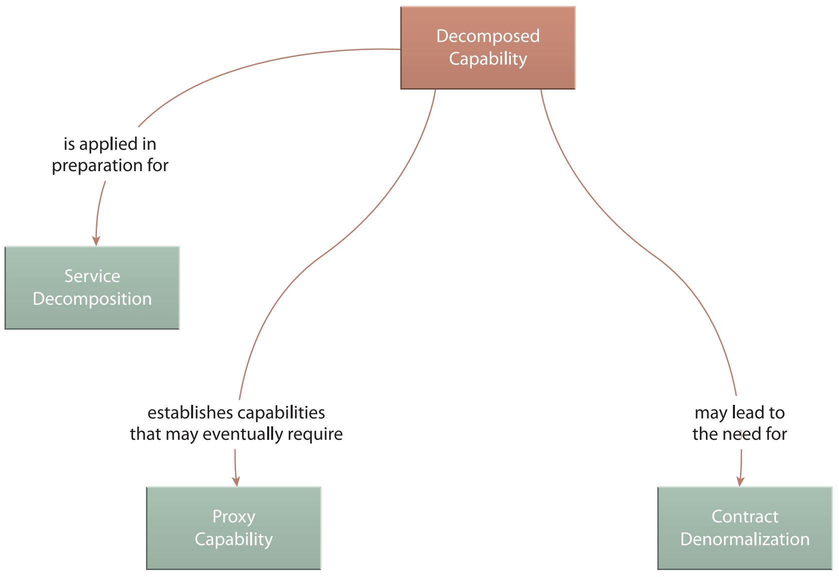


Figure 16.25
Decomposed Capability prepares a service contract for eventual decomposition, making it closely related to patterns associated with Service Decomposition (489).

CASE STUDY EXAMPLE

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

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

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

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

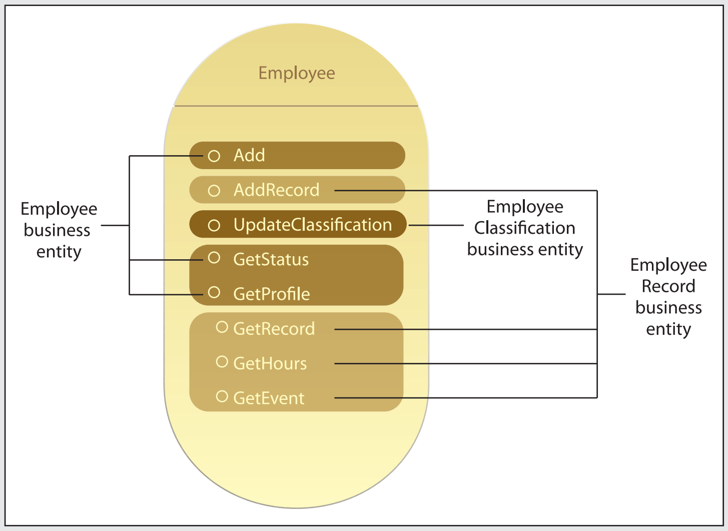


Figure 16.26
The original Employee service modeled to accommodate future decomposition by containing capabilities directly associated with known employee-related business entities. Note that not all of the capability names need to be the same as they will be when the service is decomposed into derived services.

Distributed Capability

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

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

Table 16.8
Profile summary for the Distributed Capability pattern.

Problem

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

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

019_0136135161_16.qxd 11/24/08 4:52 PM Page 511

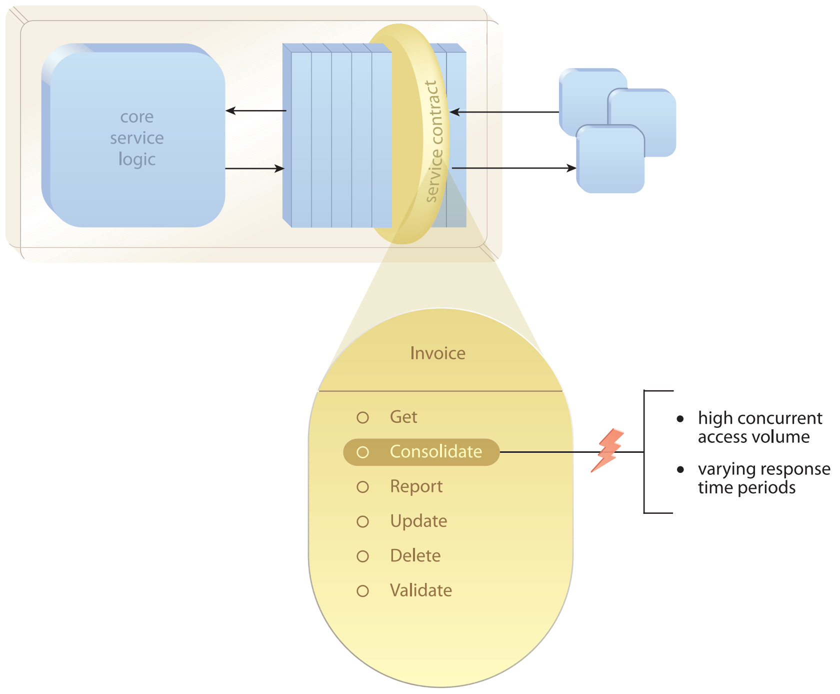


Figure 16.27
The Consolidate operation of the Invoice Web service is subject to high concurrent usage and long response periods when it is required to perform complex consolidation calculations. These factors regularly lock up server resources and therefore compromise the performance and reliability of other service operations.

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

Solution

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

019_0136135161_16.qxd 11/24/08 4:52 PM Page 512

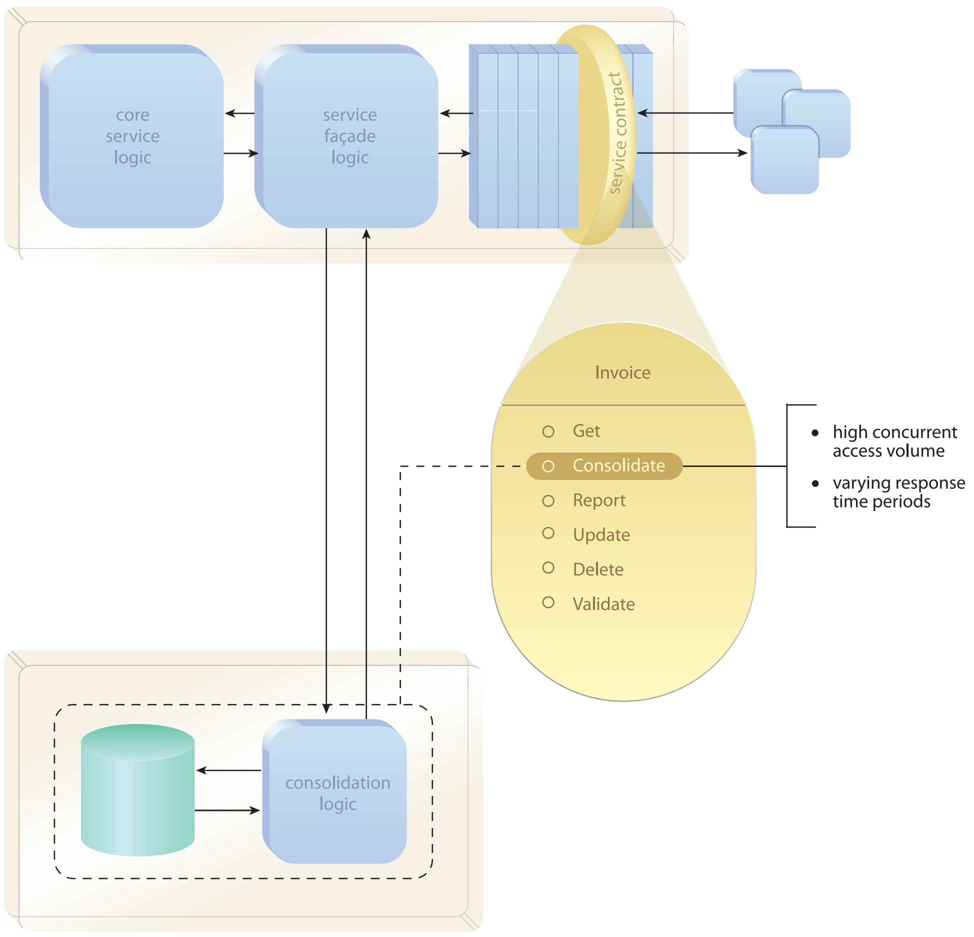


Figure 16.28

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

Application

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

Performance requirements can be somewhat streamlined by embedding additional processing logic within the façade so that it does more than just relay request and response message values. For example, the façade logic can contain routines that further parse and extract data from an incoming request message so that only the information absolutely required by the distributed capability logic is transmitted.

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

Impacts

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

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

Relationships

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

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

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

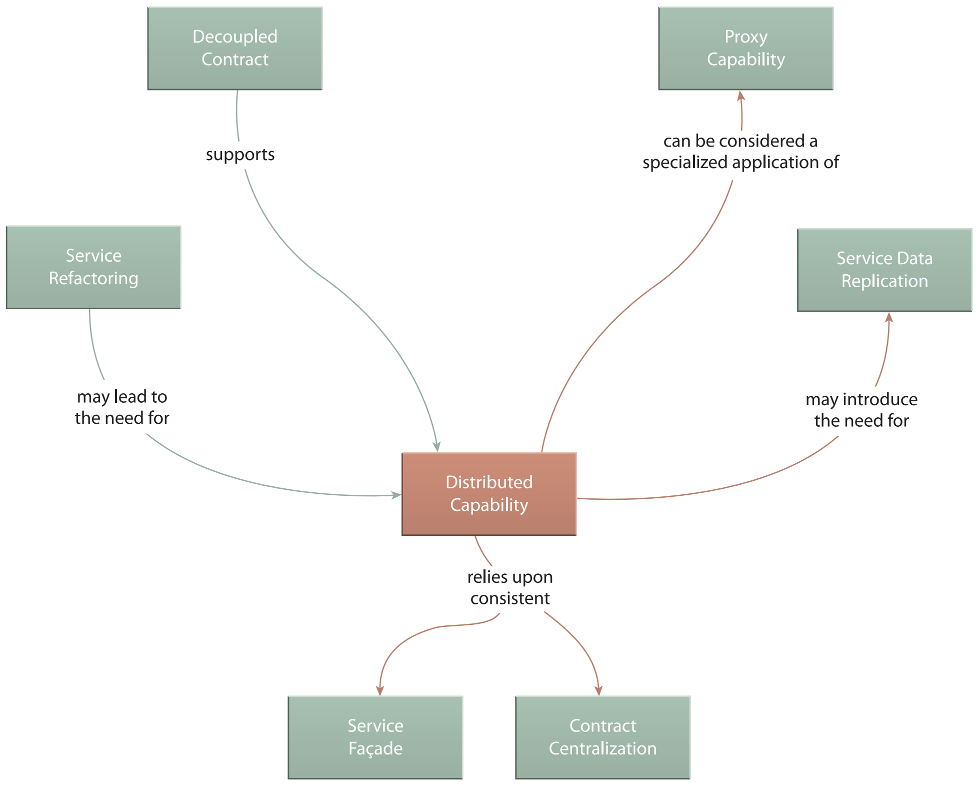


Figure 16.29

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

CASE STUDY EXAMPLE

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

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

019_0136135161_16.qxd 11/24/08 4:52 PM Page 515

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

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

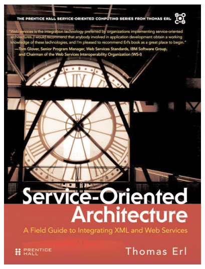
There are concerns that this one capability will tie up the service too often so that its overall scalability will decrease, thereby affecting its reliability. As a result, the team decides to separate the logic for the GetMasterReport operation to a dedicated server. The Employee Record service is equipped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

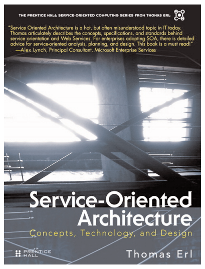
NOTE

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

019_0136135161_16.qxd 11/24/08 4:52 PM Page 516

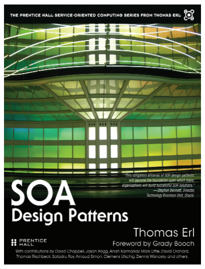
Index of Patterns


Service-Oriented Architecture
A Field Guide to Integrating XML and Web Services
Thomas Erl


Service-Oriented Architecture
Concepts, Technology, and Design
Thomas Erl

SOA
Principles of Service Design
Thomas Erl

Web Service Contract
Design & Versioning
for SOA


SOA
Design Patterns
Thomas Erl
Foreword by Grady Booch