THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

Web Service Contract Design & Versioning for SOA

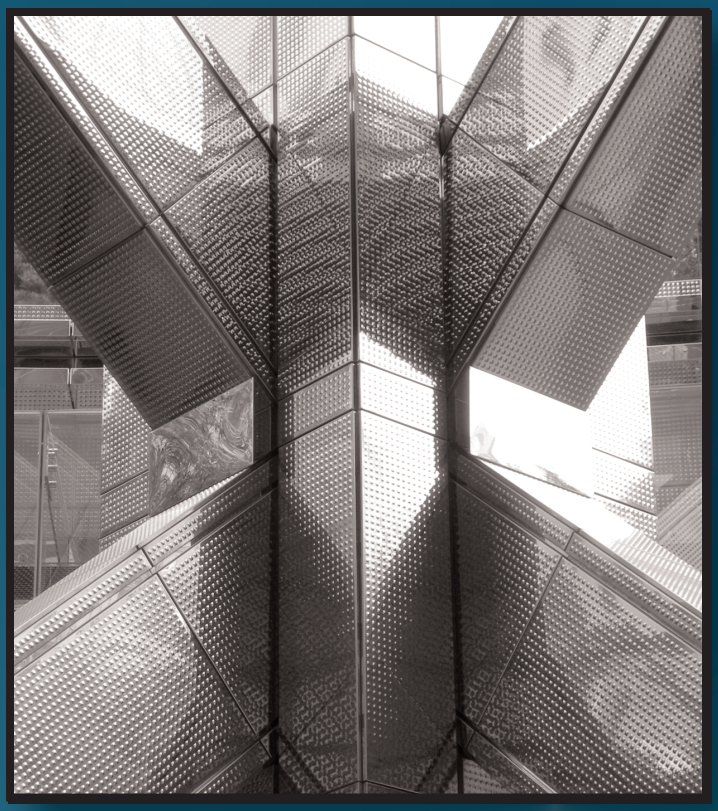
Thomas Erl, Anish Karmarkar, Priscilla Walmsley,
Hugo Haas, Umit Yalcinalp, Canyang (Kevin) Liu,
David Orchard, Andre Tost, James Pasley

PRENTICE HALL

Web Service Contract Design and Versioning for SOA

Thomas Erl, Anish Karmarkar, Priscilla Walmsley,
Hugo Haas, Umit Yalcinalp, Canyang Kevin Liu,
David Orchard, Andre Tost, James Pasley

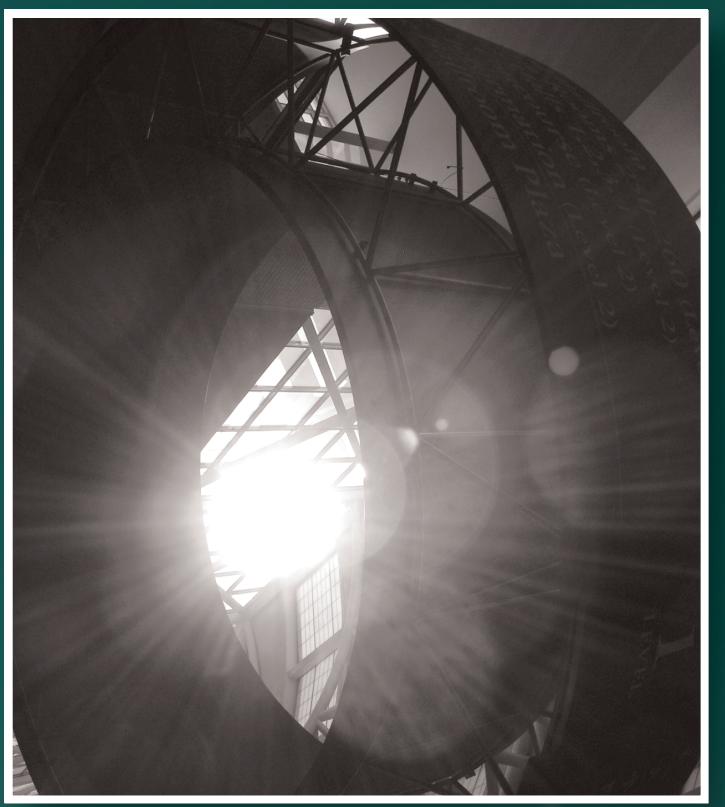
PRENTICE HALL
UPPER SADDLE RIVER, NJ • BOSTON • INDIANAPOLIS • SAN FRANCISCO
NEW YORK • TORONTO • MONTREAL • LONDON • MUNICH • PARIS • MADRID
CAPETOWN • SYDNEY • TOKYO • SINGAPORE • MEXICO CITY



Part I

Fundamental Service Contract Design

As we established in Chapter 1, this is not a book about SOA, nor is it a book about Web services. Several books have been published over the past few years that address these topics individually. What we are focused on is the design and versioning of Web service contracts in support of service-oriented solution design. What this requires us to do is explore the marriage of Web service contract technologies with the service-orientation design paradigm and the service-oriented architectural model.



Chapter 3

SOA Fundamentals and Web Service Contracts

As we established in Chapter 1, this is not a book about SOA, nor is it a book about Web services. Several books have been published over the past few years that address these topics individually. What we are focused on is the design and versioning of Web service contracts in support of service-oriented solution design. What this requires us to do is explore the marriage of Web service contract technologies with the service-orientation design paradigm and the service-oriented architectural model.

As a starting point, we first need to establish some fundamental terms and concepts associated with service-oriented computing and with an emphasis of how these terms and concepts relate to Web service contracts.

3.1 Basic SOA Terminology

This section borrows some content from SOAGlossary.com to provide the following term definitions:

- Service-Oriented Computing
- Service-Orientation
- Service-Oriented Architecture (SOA)
- Service
- Service Models
- Service Composition
- Service Inventory
- Service-Oriented Analysis
- Service Candidate
- Service-Oriented Design
- Web Service
- Service Contract
- Service-Related Granularity

If you are already an experienced SOA professional, then you might want to just skim through this part of the book. The defined terms are used here and there throughout subsequent chapters.

Service-Oriented Computing

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

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

In this book we refer primarily to the strategic goals of service-oriented computing as they tie into approaches for Web service contract design and versioning. These goals are briefly described in the *Service-Oriented Computing Goals and Web Service Contracts* section.

Service-Orientation

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

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

The service-orientation design paradigm is primarily comprised of eight specific design principles, as explained in the *Service-Orientation and Web Service Contracts* section. Several of these principles can affect the design of Web service contracts.

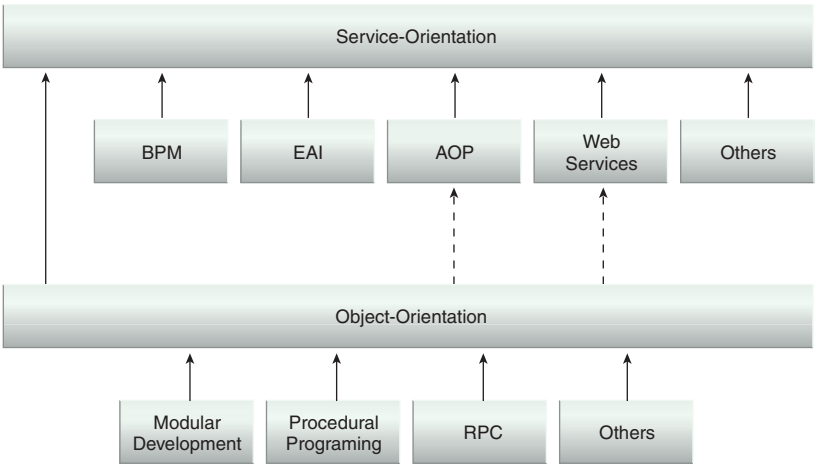


Figure 3.1
Service-orientation is very much an evolutionary design paradigm that owes much of its existence to established design practices and technology platforms.

Service-Oriented Architecture (SOA)

Service-oriented architecture represents an architectural model that aims to enhance the agility and cost-effectiveness of an enterprise while reducing the overall burden of IT on an organization. It accomplishes this by positioning services as the primary means through which solution logic is represented. SOA supports service-orientation in the realization of the strategic goals associated with service-oriented computing.

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

NOTE

Historically, the term "service-oriented architecture" (or "SOA") has been used so broadly by the media and within vendor marketing literature that it has almost become synonymous with service-oriented computing itself.

Note that the following service-oriented architecture types exist:

- Service Architecture
- Service Composition Architecture
- Service Inventory Architecture
- Service-Oriented Enterprise Architecture

As you may have guessed, we are primarily focused on the service architecture in this book. However, the decisions we make regarding the design of Web service contracts will ultimately affect the quality of related composition and inventory architectures.

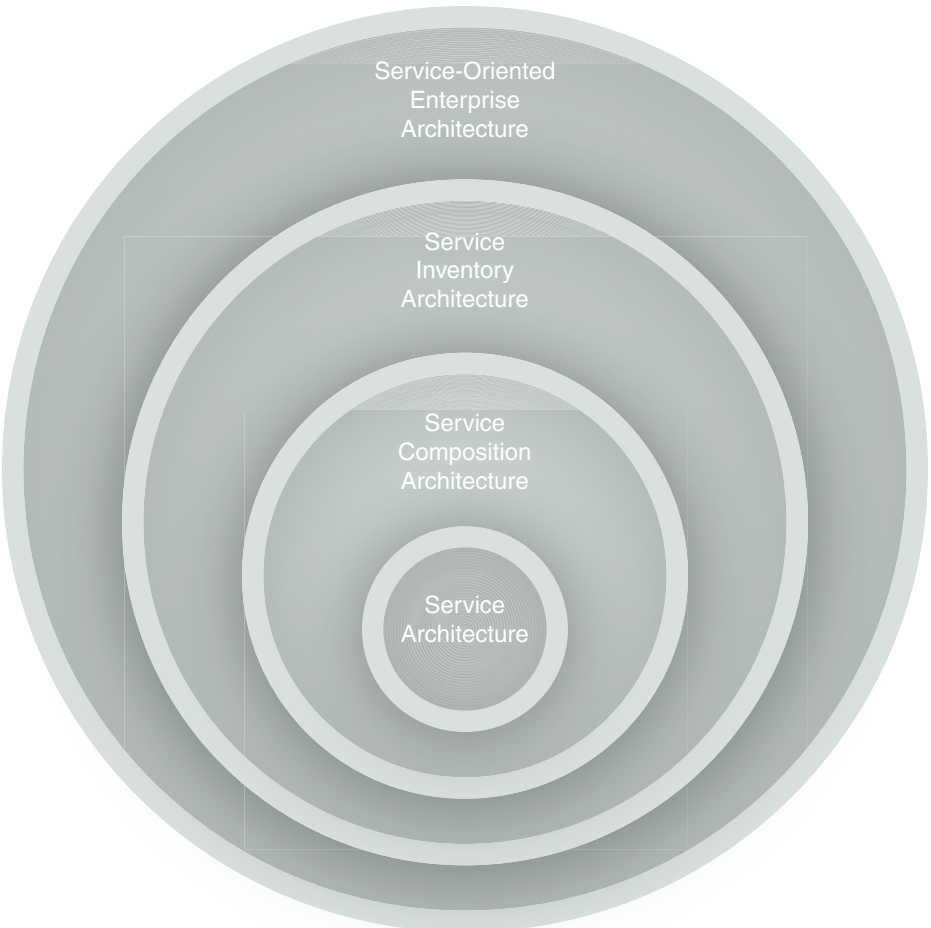


Figure 3.2
The layered SOA model that reveals how service-oriented architecture types can encompass each other. (These different architectural types are explained in the book *SOA Design Patterns*.)

Service

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

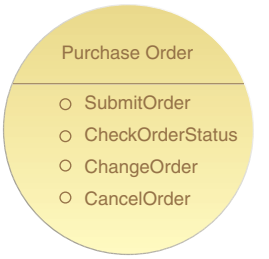


Figure 3.3
The chorded circle symbol is used to represent a service, primarily from a contract perspective.

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

Each service is assigned its own distinct functional context and is comprised of a set of capabilities related to this context. Therefore, a service can be considered a container of capabilities associated with a common purpose (or functional context). Capabilities are expressed in the service contract (defined shortly).

As we established earlier, this book is dedicated to the design of technical contracts for services built as Web services. Within a Web service contract, service capabilities are referred to as service *operations*.

Service Models

A *service model* is a classification used to indicate that a service belongs to one of several predefined types based on the nature of the logic it encapsulates, the reuse potential of this logic, and how the service may relate to domains within its enterprise.

The following three service models are common to most enterprise environments and therefore common to most SOA projects:

- 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 in order to complete its task.
- Entity Service – A reusable service with an agnostic functional context associated with one or more related business entities (such as invoice, customer, claim, etc.). For example, a Purchase Order service has a functional context associated with the processing of purchase order-related data and logic. Chapter 13 has a section dedicated to Web service contract design for entity services.

- Utility Service – Also a reusable service with an agnostic functional context, but 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.

Service models play an important role during service-oriented analysis and service-oriented design phases. Although the just listed service models are well established, it is not uncommon for an organization to create its own service models. Often these new classifications tend to be derived from one of the aforementioned fundamental service models.

NOTE

Most of the service contract examples in this book are for entity services that are required to deal with core business-related processing and the exchange of business documents.

Agnostic Logic and 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 specific to single purpose tasks is intentionally omitted, agnostic logic is considered multi-purpose. On the flipside, logic that is specific to (contains knowledge of) a single-purpose task is labeled as *non-agnostic* logic.

Another way of thinking about agnostic and non-agnostic logic is to focus on the extent to which the logic can be repurposed. Because agnostic logic is expected to be multi-purpose, it is subject to the Service Reusability principle with the intention of turning it into highly reusable logic. Once reusable, this logic is truly multi-purpose in that it, as a single software program (or service), can be used to automate multiple business processes.

Non-agnostic logic does not have these types of expectations. It is deliberately designed as a single-purpose software program (or service) and therefore has different characteristics and requirements.

Service Composition

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

Service compositions can be classified into primitive and complex variations. In early service-oriented solutions, simple logic was generally implemented via point-to-point exchanges or primitive compositions. As the surrounding technology matured, complex compositions became more common.

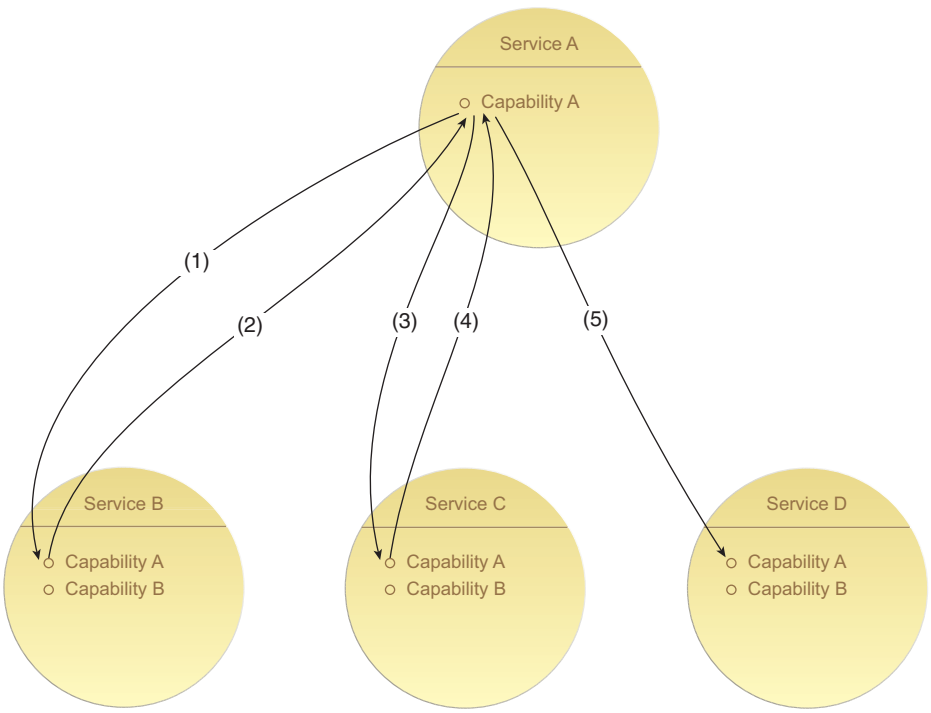


Figure 3.4
A service composition comprised of four services. The arrows indicate a sequence of modeled message exchanges. Note arrow #5 representing a one-way, asynchronous data delivery from Service A to Service D.

Much of the service-orientation design paradigm revolves around preparing services for effective participation in numerous complex compositions—so much so that the Service Composability design principle exists, dedicated solely to ensuring that services are designed in support of repeatable composition.

How service contracts are designed will influence the effectiveness and complexity potential of service compositions. Various contract-related granularity levels will determine the quantity of runtime processing and data exchange required—qualities that can end up hindering or enabling composition performance. Furthermore, techniques, such as those provided by the Contract Denormalization and Concurrent Contracts patterns,

can help optimize composition designs. (These design patterns are briefly explained at the end of this chapter in the *SOA Design Patterns and Web Service Contracts* section.)

Service Inventory

A *service inventory* is an independently standardized and governed collection of complementary services within a boundary that represents an enterprise or a meaningful segment of an enterprise. When an organization has multiple service inventories, this term is further qualified as *domain service inventory*.

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

It is worth noting that the application of the Standardized Service Contract principle is intended to be limited to the boundary of a service inventory, as are design standards-related patterns, such as Canonical Transport and Canonical Schema.

Service-Oriented Analysis

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

The service-oriented analysis process is commonly carried out iteratively, once for each business process. Typically, the delivery of a service inventory determines a scope that represents a meaningful domain or the enterprise as a whole. All iterations of a service-oriented analysis then pertain to that scope, with an end-result of a service inventory blueprint.

NOTE

Visit SOAMethodology.com for an explanation of the iterative service-oriented analysis process.

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

Service Candidate

When conceptualizing services during the service modeling sub-process of the service-oriented analysis phase, services are defined on a preliminary basis and still subject to a great deal of change and refinement before they are handed over to the service-oriented design project stage responsible for producing physical service contracts.

The term "service candidate" is used to help distinguish a conceptualized service from an actual implemented service. You'll notice a few references to service candidates in this book, especially in some of the early case study content.

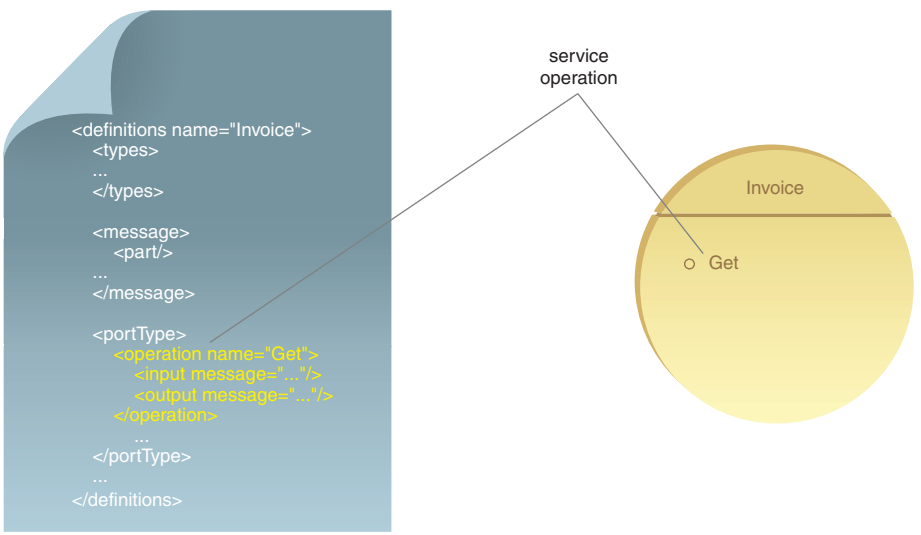


Figure 3.5

The "chorded circle" symbol (right) provides a simple representation of a service contract during both modeling and design stages. The Get operation (right) is first modeled and then forms the basis of the actual operation definition within a WSDL document (left).

Service-Oriented Design

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.

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. 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.

There is a different service-oriented design process for each of the three common service models (task, entity, and utility). The variations in process steps primarily accommodate different priorities and the nature of the logic being expressed by the contract.

This book does not discuss the process of service-oriented design in detail, but there are references to some of the considerations raised by the "contract-first" emphasis of this process.

Web Service

A *Web service* is a body of solution logic that provides a physically decoupled technical contract consisting of a WSDL definition and one or more XML Schema and/or WS-Policy definitions. As we will explore in this book, these documents can exist in one physical file or be spread across multiple files and still be part of one service contract. Spreading them out makes them reusable across multiple contracts.

The Web service contract exposes public capabilities as operations, establishing a technical interface comparable to a traditional application programming interface (API) but without any ties to proprietary communications framework.

The logic encapsulated by a Web service does not need to be customized or component-based. Legacy application logic, for example, can be exposed via Web service contracts through the use of service adapter products. When custom-developed, Web service logic is typically based on modern component technologies, such as Java and .NET. In this case, the components are further qualified as core service logic.

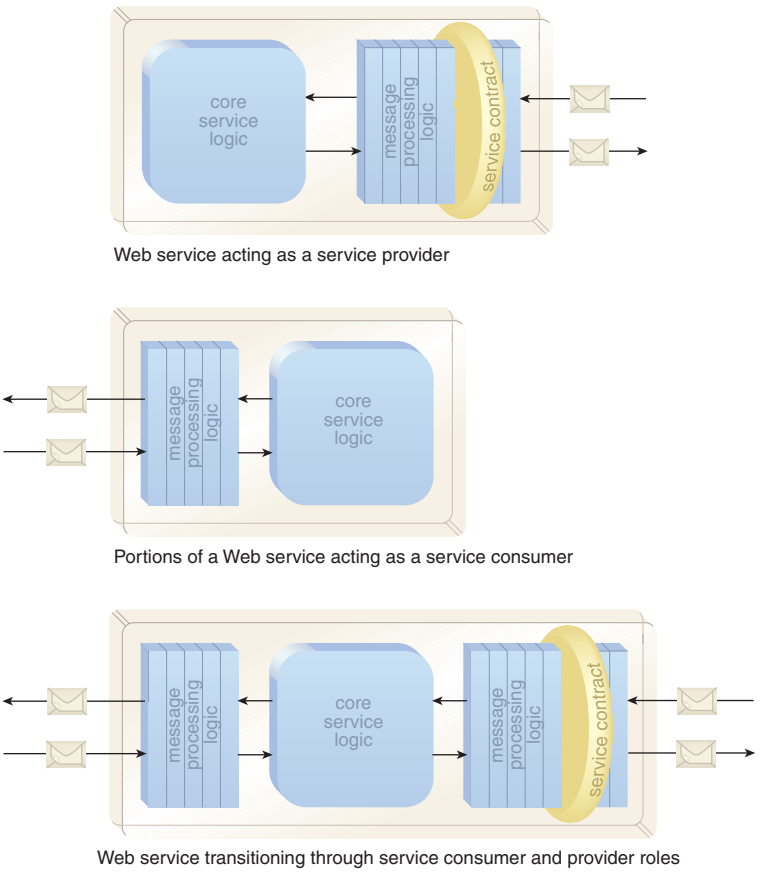


Figure 3.6

Three variations of a single Web service showing the different physical parts of its architecture that come into play, depending on the role it assumes at runtime. Note the cookie-shaped symbol that represents the service contract wedged in between layers of agent-driven message processing logic. This is the same chorded circle symbol shown earlier but from a different perspective.

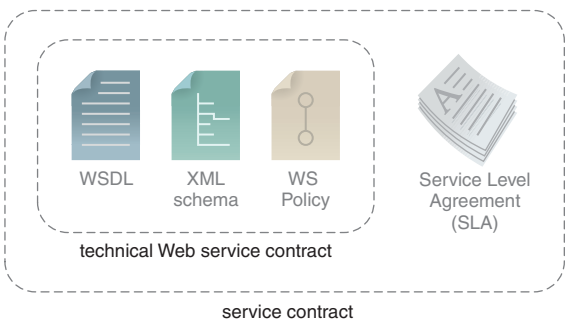
Service Contract

A *service contract* is comprised of one or more published documents that express meta information about a service. The fundamental part of a service contract consists of the documents that express its technical interface. These form the technical service contract, which essentially establishes an API into the functionality offered by the service via its capabilities.

When services are implemented as Web services, the most common service description documents are the WSDL definition, XML schema definition, and WS-Policy definition. A Web service generally has one WSDL definition, which can link to multiple XML schema and policy definitions. When services are implemented as components, the technical service contract is comprised of a technology-specific API.

A service contract can be further comprised of human-readable documents, such as a Service Level Agreement (SLA) that describes additional quality-of-service features, behaviors, and limitations. As we discuss in the WS-Policy chapters, several SLA-related requirements can also be expressed in machine-readable format as policies.

Figure 3.7
The common documents that comprise the technical Web service contract, plus a human-readable SLA.



Within service-orientation, the design of the service contract is of paramount importance—so much so, that the Standardized Service Contract design principle and the aforementioned service-oriented design process are dedicated solely to the standardized creation of service contracts.

Note that because this book is focused only on technical contracts for Web services, the terms "service contract" and "Web service contract" are used interchangeably.

Service-Related Granularity

When designing services, there are different granularity levels that need to be taken into consideration, as follows:

- Service Granularity – Represents the functional scope of a service. For example, fine-grained service granularity indicates that there is little logic associated with the service's overall functional context.
- Capability Granularity – The functional scope of individual service capabilities (operations) 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. The more coarse constraint granularity is, the less constraints (or smaller the amount of validation logic) a given capability will have.
- Data Granularity – This granularity level represents the quantity of data processed. From a Web service perspective, this corresponds to input, output, and fault messages. 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 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 a Web service contract.

In this book you will especially notice references to constraint granularity because so much of contract design relates to the definition of validation logic constraints.

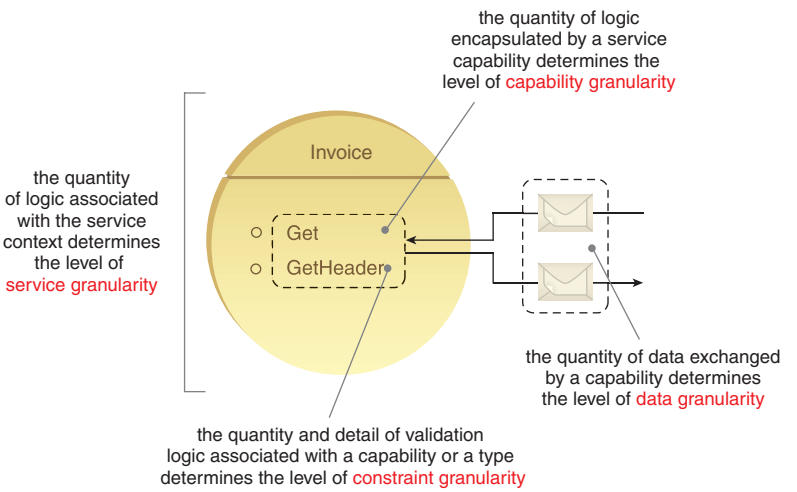


Figure 3.8

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.

Further Reading

As mentioned at the beginning of this section, all of these terms are defined at SOA-Glossary.com. More detailed explanations are available at WhatIsSOA.com and in Chapters 3 and 4 of *SOA: Principles of Service Design*. If you are not familiar with service-oriented computing, it is recommended that you read through these additional descriptions.

3.2 Service-Oriented Computing Goals and Web Service Contracts

It's always good to get an idea of the big picture before diving into the details of any technology-centric topic. For this reason, we'll take the time to briefly mention the overarching goals and benefits associated with service-oriented computing as they relate to Web service contract design.

Because these goals are strategic in nature, they are focused on long-term benefit—a consideration that ties into both the design and governance of services and their contracts. An understanding of these long-term benefits helps provide a strategic context for many of the suggested techniques and practices in this guide.

Here's the basic list of the goals and benefits of service-oriented computing:

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

Although it might not be evident, service contract design touches each of these goals to some extent.

Let's explore how.

Increased Intrinsic Interoperability

For services to attain a meaningful level of intrinsic interoperability, their technical contracts must be highly standardized and designed consistently to share common expressions and data models. This fundamental requirement is why project teams often must take control of their Web service contracts instead of allowing them to be auto-generated and derived from different sources.

Increased Federation

Service-oriented computing aims to achieve a federated service endpoint layer. It is the service contracts that are the endpoints in this layer, and it is only through their consistent and standardized design that federation can be achieved. This, again, is a goal that is supported by the ability of a project team to customize and refine Web service contracts so that they establish consistent endpoints within a given service inventory boundary.

Increased Vendor Diversification Options

For a service-oriented architecture to allow on-going vendor diversification, individual services must effectively abstract proprietary characteristics of their underlying vendor technology. The contract remains the only part of a service that is published and available to consumers. It must therefore be deliberately designed to express service capabilities without any vendor-specific details. This extent of abstraction allows service owners to extend or replace vendor technology. Vendor diversification is especially attainable through the use of Web services, due to the fact that they are supported by all primary vendors while providing a non-proprietary communications framework.

Increased Business and Technology Domain Alignment

The service layers that tend to yield the greatest gains for service-oriented environments are those comprised of business-centric services (such as task and entity services). These types of services introduce an opportunity to effectively express various forms of business logic in close alignment with how this logic is modeled and maintained by business analysts.

This expression is accomplished through service contracts and it is considered so important that entire modeling processes and approaches exist to first produce a conceptual version of the service contract prior to its physical design.

Strategic Benefits

The latter three goals listed in the previous bullet list represent strategic benefits that are achieved when attaining the first four goals. We therefore don't need to map the relevance of service contracts to each of them individually.

If we take the time to understand how central service contract design is to the ultimate target state we hope to achieve with service-oriented computing in general, it's clear to see why this book was written.

Further Reading

Formal descriptions for each of these strategic goals are available at WhatIsSOA.com and in Chapter 3 of *SOA: Principles of Service Design*. While it's good to have an understanding of these goals and benefits, it is not required to learn the technologies covered in this book.

3.3 Service-Orientation and Web Service Contracts

To understand SOA is to understand service-orientation, the design paradigm that establishes what is required in order to create software programs that are truly service-oriented.

Service-orientation represents a design approach comprised of eight specific design principles. Service contracts tie into most but not all of these principles. Let's first introduce their official definitions:

- Standardized Service Contract – "Services within the same service inventory are in compliance with the same contract design standards."
- Service Loose Coupling – "Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment."
- Service Abstraction – "Service contracts only contain essential information and information about services is limited to what is published in service contracts."
- Service Reusability – "Services contain and express agnostic logic and can be positioned as reusable enterprise resources."
- Service Autonomy – "Services exercise a high level of control over their underlying runtime execution environment."

- Service Statelessness – "Services minimize resource consumption by deferring the management of state information when necessary."
- Service Discoverability – "Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted."
- Service Composability – "Services are effective composition participants, regardless of the size and complexity of the composition."

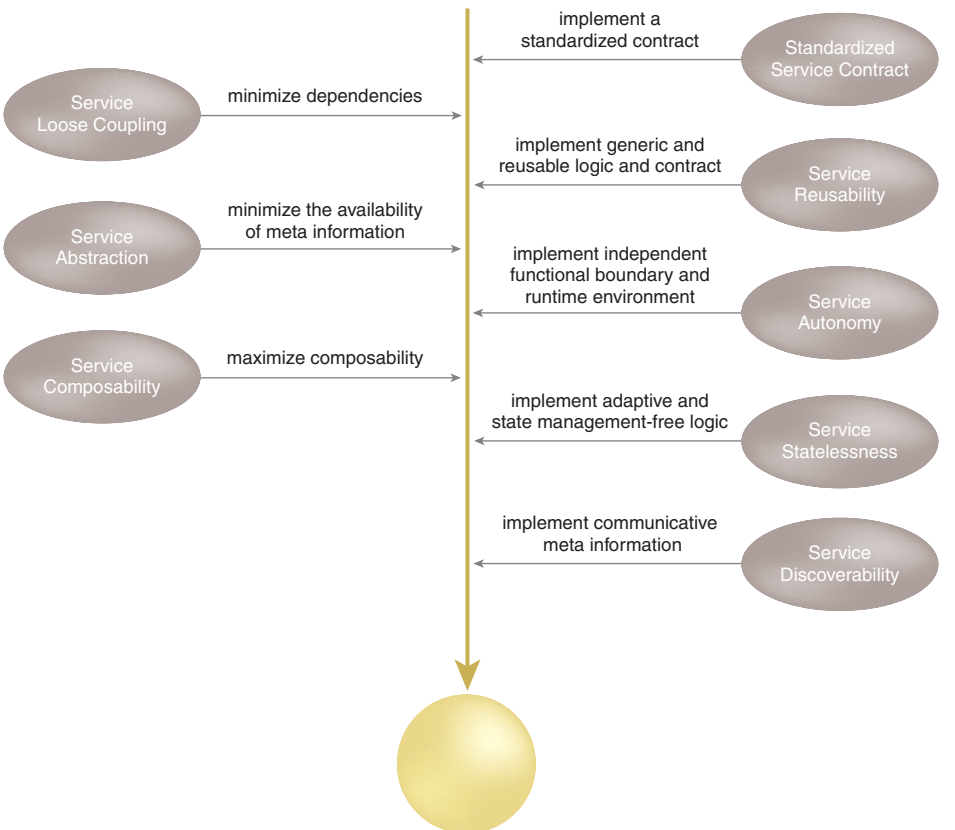


Figure 3.9
How service-orientation design principles can collectively shape service design.

Each of these design principles can, to some extent, influence how we decide to build a Web service contract. With regards to the topics covered in this book, the following principles have a direct impact.

Standardized Service Contract

Given its name, it's quite evident that this design principle is only about service contracts and the requirement for them to be consistently standardized within the boundary of a service inventory. This design principle essentially advocates "contract first" design for services.

Service Loose Coupling

This principle also relates to the service contract. Its design and how it is architecturally positioned within the service architecture are regulated with a strong emphasis on ensuring that only the right type of content makes its way into the contract in order to avoid the negative coupling types.

The following sections briefly describe common types of coupling. All are considered negative coupling types, except for the last.

Contract-to-Functional Coupling

Service contracts can become dependent on outside business processes, especially when they are coupled to logic that was designed directly in support of these processes. This can result in contract-to-functional coupling whereby the contract expresses characteristics that are specifically related to the parent process logic.

Contract-to-Implementation Coupling

When details about a service's underlying implementation are embedded within a service contract, an extent of contract-to-implementation coupling is formed. This negative coupling type commonly results when service contracts are a native part of the service implementation (as with component APIs) or when they are auto-generated and derived from implementation resources, such as legacy APIs, components, and databases.

Contract-to-Logic Coupling

The extent to which a service contract is bound to the underlying service programming logic is referred to as contract-to-logic coupling. This is considered a negative type of service coupling because service consumer programs that bind to the service contract end up also inadvertently forming dependencies on the underlying service logic.

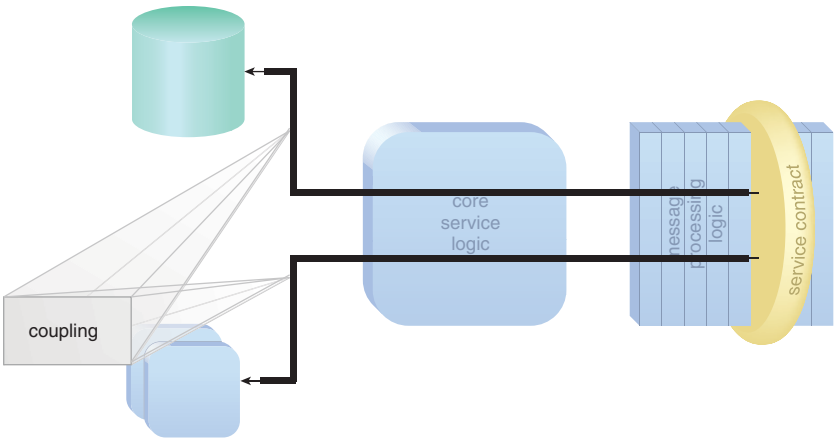


Figure 3.10
A Web service contract can be negatively coupled to various parts of the underlying service implementation.

Contract-to-Technology Coupling

When the contract exposed by a service is bound to non-industry-standard communications technology, it forms an extent of contract-to-technology coupling. Although this coupling type could be applied to the dependencies associated with any proprietary technology, it is used exclusively for communications technology because that is what service contracts are generally concerned with.

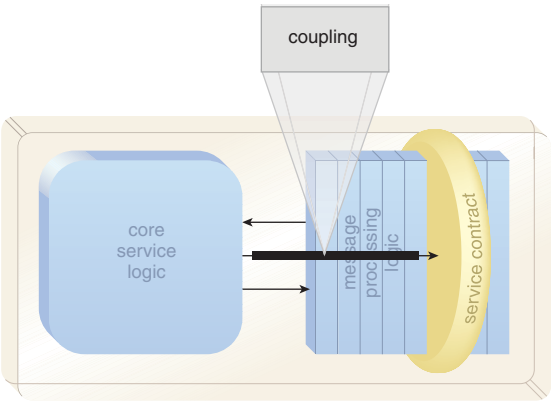
An example of contract-to-technology coupling is when the service exists as a distributed component that requires the use of a proprietary RPC technology. Because this book is focused solely on Web service contract technology, this coupling type does not pose a design concern.

Logic-to-Contract Coupling

Each of the previously described forms of coupling are considered negative because they can shorten the lifespan of a Web service contract, thereby leading to increased governance burden as a result of having to manage service contract versions.

This book is focused on providing the skills necessary to achieve high levels of *logic-to-contract* coupling by ensuring that the Web service contract can be designed with complete independence from the underlying Web service implementation.

Figure 3.11
The most desirable design is for the Web service contract to remain an independent and fully decoupled part of the service architecture, thereby requiring the underlying logic to be coupled to it.



Service Abstraction

By turning services into black boxes, the contracts are all that is officially made available to consumer designers who want to use the services. While much of this principle is about the controlled hiding of information by service owners, it also advocates the streamlining of contract content to ensure that only essential content is made available. The related use of the Validation Abstraction pattern further can affect aspects of contract design, especially related to the constraint granularity of service capabilities.

Service Reusability

While this design principle is certainly focused on ensuring that service logic is designed to be robust and generic and much like a commercial product, these qualities also carry over into contract design. When viewing the service as a product and its contract as a generic API to which potentially many consumer programs will need to interface, the requirement emerges to ensure that the service's functional context, the definition of its capabilities, and the level at which each of its design granularities are set are appropriate for it to be positioned as a reusable enterprise resource.

Service Discoverability

Because the service contracts usually represent all that is made available about a service, they are what this principle is primarily focused on when attempting to make each service as discoverable and interpretable as possible by a range of project team members.

Note that although Web service contracts need to be designed to be discoverable, this book does not discuss discovery processes or registry-based architectures.

Service Composability

This regulatory design principle is very concerned with ensuring that service contracts are designed to represent and enable services to be effective composition participants. The contracts must therefore adhere to the requirements of the previously listed design principles and also take multiple and complex service composition requirements into account.

Further Reading

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

3.4 SOA Design Patterns and Web Service Contracts

Design patterns provide proven solutions to common design problems. SOA has matured to an extent where a catalog of design patterns has been established. Of interest to us are those that affect the design and versioning of service contracts, specifically:

- *Canonical Expression* – Service contracts are standardized using naming conventions.
- *Canonical Schema* – Schema data models for common information sets are standardized across service contracts within a service inventory boundary.
- *Canonical Versioning* – Service contracts within the same service inventory are subject to the same versioning rules and conventions.
- *Concurrent Contracts* – Multiple contracts can be created for a single service, each targeted at a specific type of consumer.
- *Compatible Change*—Already implemented service contracts are revised without breaking backwards compatibility.
- *Contract Centralization* – Access to service logic is limited to the service contract, forcing consumers to avoid negative contract-to-implementation coupling.
- *Contract Denormalization* – 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.

- *Decomposed Capability* – Services prone to future decomposition can be equipped with a series of granular capabilities that more easily facilitate decomposition.
- *Decoupled Contract* – The service contract is physically decoupled from its implementation.
- *Distributed Capability* – 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.
- *Messaging Metadata* – The message contents can be supplemented with activity-specific metadata that can be interpreted and processed separately at runtime.
- *Partial Validation* – Service consumers are designed to validate a subset of the data received from a service.
- *Policy Centralization* – Global or domain-specific policy assertions can be isolated and applied to multiple services.
- *Proxy Capability* – When a service contract needs to be decomposed, the original service contract can be preserved, even if underlying capability logic is separated, by turning the established capability definition into a proxy.
- *Schema Centralization* – Select schemas that exist as physically separate parts of the service contract are shared across multiple contracts.
- *Service Messaging* – Services can be designed to interact via a messaging-based technology, which removes the need for persistent connections and reduces coupling requirements.
- *Termination Notification* – Service contracts are extended to express termination information.
- *Validation Abstraction* – Granular validation logic and rules can be abstracted away from the service contract, thereby decreasing constraint granularity and increasing the contract's potential longevity.
- *Version Identification* – Version numbers and related information is expressed within service contracts.

Web Services and the Decoupled Contract Pattern

It is worth singling out Decoupled Contract at this stage because a Web service contract is essentially an implementation of this pattern. When building services as Web services, service contracts are positioned as physically separate parts of the service architecture. This allows us to fully leverage the technologies covered in this book in order to design and develop these contracts independently from the logic and implementations they will eventually represent.

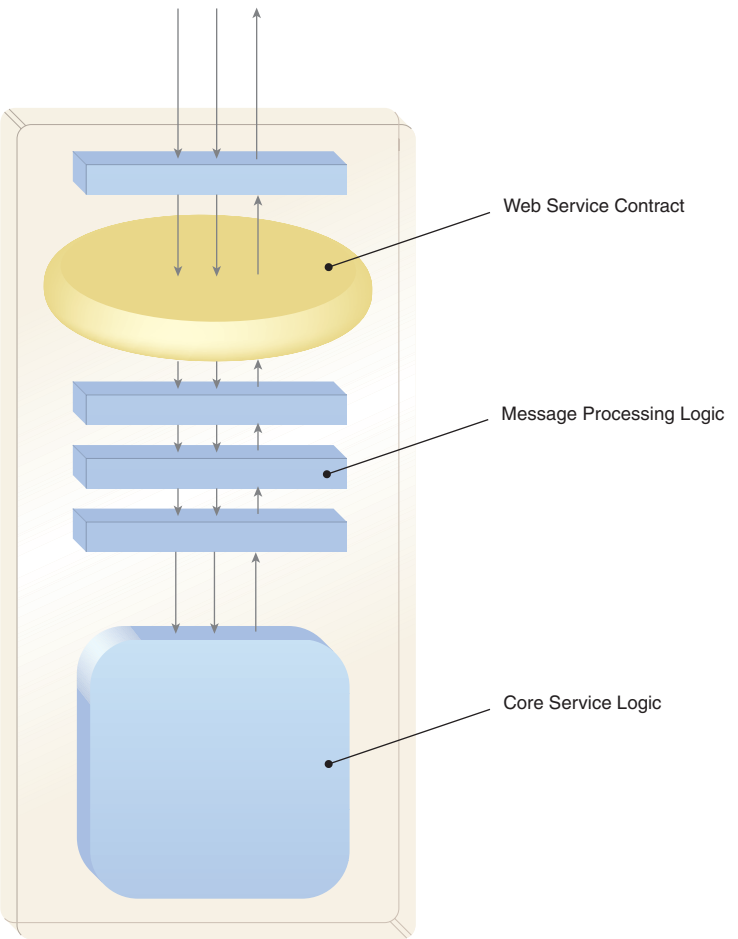


Figure 3.12
The Web service contract is a physically separated part of a Web service implementation.

Therefore, you may not see a lot of references to the Decoupled Contract pattern because it goes without saying that a Web service contract is naturally decoupled. However, it is always important to keep the coupling types explained earlier in the *Service-Orientation and Web Service Contracts* section in mind because although physically decoupled, the content of any Web service contract can still be negatively coupled to various parts of the service environment.

Further Reading

The previously listed design patterns are part of a larger design pattern catalog published in the book *SOA Design Patterns*. Concise descriptions of these patterns are also available at SOAPatterns.org.