Oracle® Reference Architecture

SOA Foundation

Release 3.2

E14484-03

February 2012

ORACLE®

ORA SOA Foundation, Release 3.2

E14484-03

Primary Author: Stephen G. Bennett, Mark Wilkins

Contributing Author: Dave Chappelle, Bob Hensle, Anbu Krishnaswamy, Cliff Booth, Jeff McDaniel

Contributor: Clemens Utschig-Utschig, David Gresham, Sameer Vaishampayan, Martin Cookson, Daniel Healy

Contents

4 Technologies and Standards

5 Summary

A Other Service Models

List of Figures

Send Us Your Comments

ORA SOA Foundation, Release 3.2

E14484-03

Oracle welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this document?

If you find any errors or have any other suggestions for improvement, please indicate the title and part number of the documentation and the chapter, section, and page number (if available). You can send comments to us at its_feedback_ww@oracle.com.

Preface

This document is part of a series of documents that describe the Oracle Reference Architecture (ORA) Service Oriented Architecture (SOA) strategy. This “SOA Foundation" document provides the underlying architectural definitions of associated SOA concepts. Contained within this document are background and definitions of SOA along with detailed definitions of a Service. The SOA Foundation document presents important basic concepts of SOA that are fundamental to building applications for a SOA environment. It covers topics including the components of a Service, Service layering, Service types, the Service model, composite applications, architectural styles for SOA implementation, invocation patterns, and standards that apply to SOA. Benefits and architectural principles are identified throughout while various implementation strategies are also considered.

The main focus of this document is to address the challenges faced by architects attempting SOA, by explaining the underpinning architectural concepts that distinguish SOA from architectural approaches.

Document Purpose

The purpose of this document is to describe the foundational aspects of SOA in support of the broader Oracle Reference Architecture. This document is intended to provide current context for SOA so the reader will understand SOA fundamentals underpinning the ORA. Various architectural approaches to SOA are described along with corresponding standards, where they exist. This document also introduces a Service model to define unambiguously what we mean by a "Service" and potentially as a starting point for use within a broader engineering modeling activity.

This SOA Foundation document presents important basic concepts of SOA that are instrumental to building applications for a SOA environment. It covers topics including the components of a Service, Service layering, Service types, the Service model, composite applications, invocation patterns, and standards that apply to SOA.

Audience

This document is intended for Service and process architects who want to understand the best way to architect applications, processes, and Services for the Oracle Fusion environment. It may also be used by Enterprise Architects to help formulate reference architectures for specific customer environments.

Document Structure

This document is organized into chapters intended to introduce SOA in the context of ORA.

Chapter 1 - Offers an overview of what SOA is and its benefits.

Chapter 2 - Defines the key characteristics of service as well as introducing a formal definition. In addition, a UML meta-model for a Service is highlighted.

Chapter 3 - Define a conceptual view of SOA, including architecture principles and key capabilities

Chapter 4 - Explores standards and technologies.

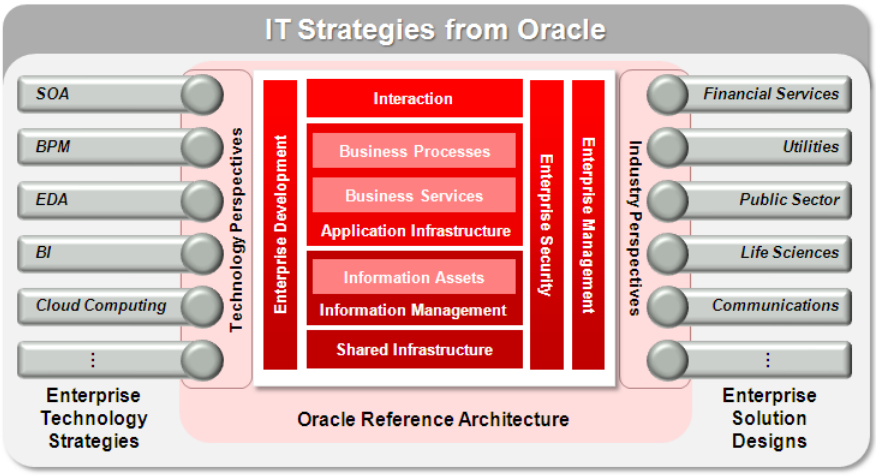
How to Use This Document

This document should be read by anyone who is interested in learning about or leveraging the Oracle Reference Architecture. It is the foundational document providing SOA context within the collection of documents that comprise ORA.

For those already familiar with the core principles of SOA, Chapter 1 can be skimmed. Chapters 2 and 3 should be read by all types of architects in order to gain an understanding of the Oracle definitions of SOA, its various architectural approaches, and the foundation necessary to realize a successful SOA strategy. Chapter 4 should be read by architects who have an interest in deployment approaches for SOA.

Related Documents

IT Strategies from Oracle (ITSO) is a series of documentation and supporting collateral designed to enable organizations to develop an architecture-centric approach to enterprise-class IT initiatives. ITSO presents successful technology strategies and solution designs by defining universally adopted architecture concepts, principles, guidelines, standards, and patterns.



ITSO is made up of three primary elements:

- **Oracle Reference Architecture (ORA)** defines a detailed and consistent architecture for developing and integrating solutions based on Oracle technologies. The reference architecture offers architecture principles and guidance based on recommendations from technical experts across Oracle. It covers a broad

spectrum of concerns pertaining to technology architecture, including middleware, database, hardware, processes, and services.

- **Enterprise Technology Strategies (ETS)** offer valuable guidance on the adoption of horizontal technologies for the enterprise. They explain how to successfully execute on a strategy by addressing concerns pertaining to architecture, technology, engineering, strategy, and governance. An organization can use this material to measure their maturity, develop their strategy, and achieve greater levels of success and adoption. In addition, each ETS extends the Oracle Reference Architecture by adding the unique capabilities and components provided by that particular technology. It offers a horizontal technology-based perspective of ORA.
- **Enterprise Solution Designs (ESD)** are industry specific solution perspectives based on ORA. They define the high level business processes and functions, and the software capabilities in an underlying technology infrastructure that are required to build enterprise-wide industry solutions. ESDs also map the relevant application and technology products against solutions to illustrate how capabilities in Oracle's complete integrated stack can best meet the business, technical, and quality of service requirements within a particular industry.

ORA SOA Foundation, along with ORA SOA Infrastructure, extend the Oracle Reference Architecture. They are part of a series of documents that comprise the SOA Enterprise Technology Strategy, which is included in the IT Strategies from Oracle collection.

Please consult the ITSO web site for a complete listing of SOA and ORA documents as well as other materials in the ITSO series.

Conventions

The following typeface conventions are used in this document:

Convention	Meaning
boldface text	Boldface type in text indicates a term defined in the text, the ***ORA Master Glossary***, or in both locations.
italic text	Italics type in text indicates the name of a document or external reference.
underline text	Underline text indicates a hypertext link.

In addition, the following conventions are used throughout the SOA documentation:

"Service" v. "service" - In order to distinguish the "Service" of Service Oriented Architecture, referred to throughout the SOA ETS, the word appears with its initial letter capitalized ("Service"), while all other uses of the word appear in all lower-case (e.g. "telephone service"); exceptions to this rule arise only when the word "service" is part of a name, such as, "Java Message Service" ("JMS"), "Web Service", etc.

1 Introduction

Service Oriented Architecture (SOA) is a strategy for constructing business-focused, software systems from loosely coupled, interoperable building blocks (called Services) that can be combined and reused quickly, within and between enterprises, to meet business needs.

Within the scope of the architectural strategy, SOA describes an approach for enterprise systems development and integration that is both technology agnostic (insofar as it operates across heterogeneous systems) and aligned with business imperatives; it provides loose coupling of course-grained components (Services) for rapid and effective reuse of enterprise IT assets.

The ultimate goal of SOA is to facilitate the creation of new business solutions through the composition of Services without the need for complex programmatic code development that might otherwise duplicate existing capabilities. This leads to a more agile and efficient enterprise that can respond more rapidly to changing market and regulatory demands.

SOA is also an architecture driven design discipline conceived to achieve the goals of increased inter-operability (information exchange, reusability, and "compose-ability"), increased federation (uniting resources and applications while maintaining their individual autonomy and self-governance), and increased business and technology domain alignment.

As a strategy SOA is broader than merely a Reference Architecture (RA) and should encompass engineering practices, governance, and other aspects of an enterprise engineering activity. SOA is an Enterprise Architectural (EA) strategy, but it is not a substitute for Enterprise Architecture: SOA is not intended to address all the concerns of a complete EA. For example, a complete EA should be concerned with network intrusion detection, desktop platform standards, and while SOA Reference Architecture also addresses concerns in the areas of security or standards, it does so only insofar as they pertain to Services.

1.1 SOA in Today's Context

SOA addresses a number of major concerns that have developed throughout the Information Technology (IT) industry over many years. The following list offers a summary of those issues for the purpose of establishing the context for SOA in today's IT environment.

- **Heterogeneity**: technology independence and interoperability (encompassing application integration). Integration was previously associated with techniques such as Message Oriented Middleware (MOM) and Enterprise Application Integration (EAI), but these suffered primarily from a need to create adapters for

every technology and application interface: this issue is addressed by SOA through its technology independent interface definition approach.

- **Business focus**: just as many businesses realized the need to become customer oriented, IT has realized the need to become business focused. Previous attempts to match business expectations focused on requirements management and software engineering methods, but SOA takes a significant step further by establishing a Service contract and subsequent encapsulation of autonomous business function, process, and/or data by the Service.
- **Agility, reuse, and cost reduction** are not only important business drivers for IT, they have also been significant problems for IT for many years. Historically, application development projects spanned many years. Today businesses expect results in far shorter timeframes and for much lower cost. Once SOA is established across an enterprise, its strategies for composition (and reuse) result in greater flexibility and faster solution delivery

SOA views functional decomposition from the enterprise level rather than purely from the application project perspective, avoiding the "technology first" approach. Furthermore, SOA demands inter-operability and its implementations meet this goal more effectively than any previous approach. Although not automatically achieved through its implementations, the architectural strategy provides for Service discovery and mediation which lead to loose coupling, ultimately facilitating business-level Service reuse and composition.

While SOA tackles all these major concerns, it is important to note that it is not a panacea, nor is it a replacement for the many foregoing technologies and strategies. In some respects the underlying concepts have been around since the early days of distributed computing, for example in the form of CORBA, EJB, or TUXEDO. SOA is the culmination of many years experience and should be regarded as an architectural evolution rather than a revolution.

1.2 Purpose and Benefits of SOA

This section is a summary of the purpose and benefits of SOA and, as such, uses many terms that are yet to be defined. If the terms used here are unfamiliar please refer to relevant chapters (or the *ORA Glossary*).

The intent of SOA is to provide common reusable Services that can be leveraged by a variety of consumers. These Services will typically originate from functionality and data that already exist in the enterprise - Services created by "Service-enabling" legacy assets. As new projects are implemented, standalone Services may also emerge as autonomous entities that do not have dependencies on legacy systems.

Figure 1–1 SOA Perspectives

As shown in Figure 1–1 above, SOA acts as a value-add layer on top of existing IT assets. It exposes these assets, as well as custom-built capabilities, as Services. SOA also includes infrastructure to aid in the discovery, management, mediation, monitoring, and security of Services. Services are made available to various types of Servicec consumers in order to rationalize the way business functions are performed and enterprise data is managed.

Of the many benefits of SOA agility, reuse, and integration are generally considered to the most significant.

1. Agility: SOA helps IT respond more quickly and cost-effectively to changing business needs which, in turn, helps the business respond to market conditions faster.
2. Reuse: The SOA style of architecture promotes more effective reuse at the macro (business unit of work) level rather than micro (code module or class) level.
3. SOA can also simplify interconnection to, and between, existing IT assets; be it integration with partners and customers, or simply integration of internal applications.

A list of benefits that can be expected from SOA is partly enterprise and/or industry specific but typically includes some or all of the following:

- Modular, reusable Services
- Reduce development time
- Allows development to concentrate on business capability changes
- Loose coupling
- Common standards-based infrastructure
- Open standards
- Abstraction
- Lower risk through proven technologies
- Lower defects through component reuse
- Reduced mean time to repair
- Leverage existing technologies, resources, and skill sets
- Ease of orchestration, more dynamic integration
- Reduced Total Cost of Ownership (TCO) over time

Many of these may be associated with overlapping concerns; however some redundancy is often necessary to address the needs of different stakeholders.

2

Concepts

This chapter describes some of the key concepts that pertain to SOA.

2.1 Use of the word "Service"

Even within IT the word "service" can mean many different things, so when we use this overloaded term in the name of our architectural strategy, Service-Oriented Architecture, it becomes very important to distinguish our Service from the many other uses of the word. When reading about Services in Service-Oriented Architecture it is common to find terms like "Shared Service" or "SOA Service" used to differentiate the subject of the discussion; unfortunately, these terms are all flawed for one reason or another (e.g. Services are not *always* "shared", while "SOA Service" suffers from redundancy).

For all the reasons listed above, and as already stated in the preface of this document, the Service-Oriented Architecture "Service" referred to throughout the SOA documentation is distinguished simply by the capitalization of its initial letter i.e. "Service": all other uses of the word appear in lower-case ("service") except when it is part of a name, such as, "Java Message Service" ("JMS"), "Web Service", etc.

One of the key things to get right in any SOA initiative is a clear and consistent definition of a Service and associated corporate governance to support that definition. This document provides a framework for such a definition of a Service.

2.2 Definition of a Service

The IT industry is not in complete agreement on what constitutes a Service so the following paragraphs describe Oracle's complete and unambiguous definition of a Service. The following chapters provide further details, including a formal definition of a Service using standard modeling techniques.

A Service is a means of packaging reusable software building blocks to provide functionality to users, applications, or other Services; it is an independent, self-sufficient, functional unit of work that is discoverable, manageable, measurable, has the ability to be versioned, and offers functionality that is required by a set of consumers. A Service may be shared, which means that the function offered by the Service is intended for multiple consumers, some known, and others that have not yet been identified.

A Service is comprised of three parts: the implementation (deployed code and configuration of infrastructure), the interface (means by which the Service is invoked), and the contract (a description of what the Service provides and its constraints). A useful metaphor for the breakdown of a Service in this way can be seen in the example of a power company, in which the implementation would be the methods used to

generate power, e.g., coal, solar, nuclear, etc., and the distribution grid that makes the power widely available. Users of power do not need to know how it is generated or even where it comes from - they simply need to be able to interface with the grid. The interface for power is the outlet. In order to enable many types of consumers (many different devices) to use electrical power, a standard outlet configuration and voltage specification has been established. Finally, a consumer must enter into a contract with the power company. The contract in this case is the agreement to pay for electricity at a certain rate, on a certain schedule. The power company and consumer may negotiate quality of service (QoS) and service level agreements (SLA) as part of the contract for certain users of power.

Service implementation can take many forms including J2EE, PL/SQL,.NET, mainframe code, etc. Like power generation, the decision of which technology to use should not affect the users of the Service. The technology decision should be made based on factors such as platform capability, developer expertise, and cost of ownership. In this way SOA helps IT by abstracting the users from underlying implementation details.

More important to consumers than implementation is the interface technology. SOA, though not tightly bound to Web Services technologies, does provide the most benefits when standard protocols are used. The most widely adaptable interface language available today is eXtensible Markup Language (XML). It provides an extensible data representation and, coupled with XML Schema, includes self describing metadata. Since all data is represented in standard string form, it is readily consumable by most computing platforms.

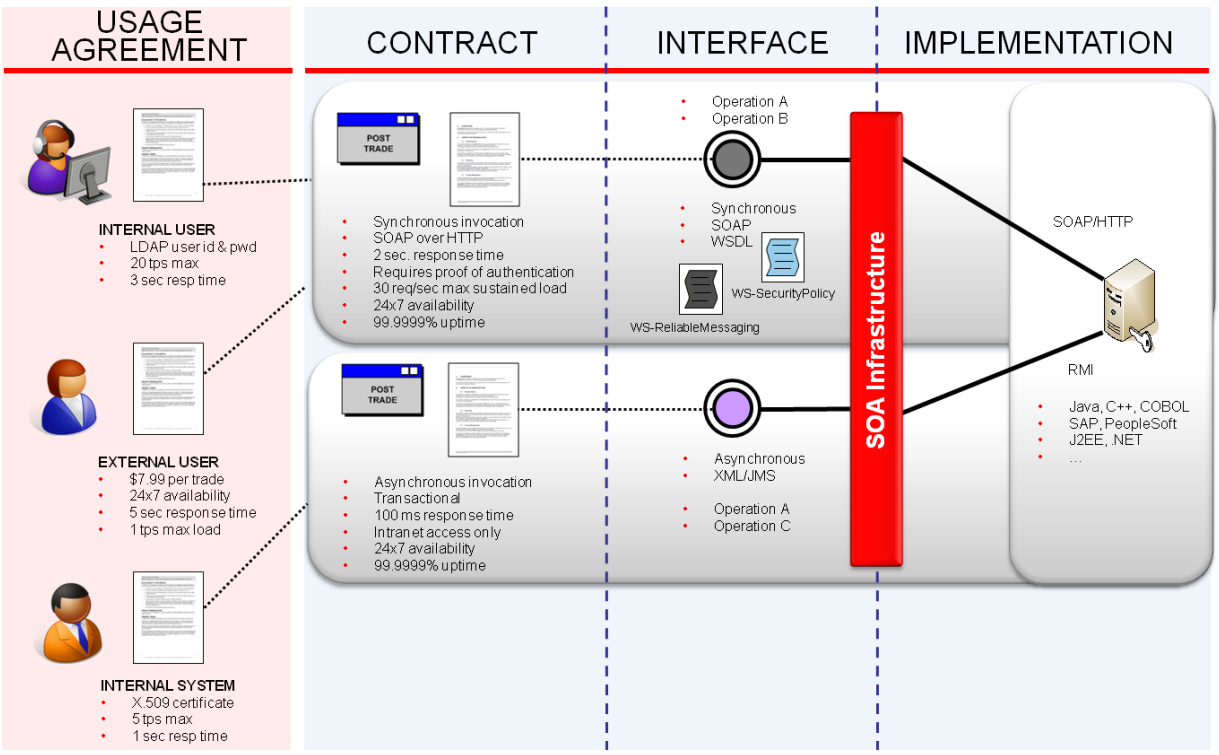
In addition to request and response data, Service interfaces may need to support additional information, such as security data, endpoint information, and binary attachments. They may also need to support certain quality of service constructs such as reliable messaging and secure conversations. For this reason request and response data should be packaged into a structure that includes headers, attachments, and is versatile enough for use by Service infrastructure. The SOAP standard provides this capability, and has become the foundation of Web Services today. Most Service infrastructure, and the numerous features they provide, are based on SOAP messaging. It is important to stress however, that SOAP and Web Services are not prerequisites for SOA and the creation of Services.

Interfaces may also support multiple transport protocols, such as HTTP, JMS, FTP, as well as interaction schemes like synchronous, asynchronous, and one-way messaging. SOAP may be transported in these various ways in order to meet quality of Service requirements.

Contracts are a way to describe functional and nonfunctional characteristics of a Service in human readable form. There is no standard representation of a contract, which means that companies and individual organizations within a company are capable of their own definitions. A contract should always be included with a Service, even though it may not contain much information. At a minimum it should describe the functional capability of the Service, what the pre- and post-conditions are, the expected performance characteristics, and security requirements.

Services may offer multiple interfaces and contracts depending on usage scenarios. For example, a Service may be offered via a synchronous interface for an online application, and an asynchronous interface to support batch processing. Each interface would have its own contract, but they could share a common implementation. This is illustrated in Figure 2–1.

Figure 2–1 Service Illustration



In addition to the service contact, interface, and implementation discussed so far, Figure 2–1 also illustrates the service consumer usage agreements. The usage agreement is not part of the Service; rather it defines what a particular service consumer is entitled to consume from the Service.

For example, suppose the service contract states that the Service guarantees to provide ten transactions per second (TPS) to the SOA environment. That clearly does not mean that each service consumer is entitled to ten TPS. Each service consumer would be limited to some portion of the overall service capability e.g. no more than three TPS.

Having both a usage agreement and a service contract provides a decoupling between the service provider and service consumer. This not only facilitates reuse but also provides a separation of concerns. The service contract defines the totality of what the Service guarantees to provide, and can be written and validated independent of any knowledge of specific service consumers. The usage agreement is service consumer specific and defines what capabilities of the Service each consumer is allowed to consume.

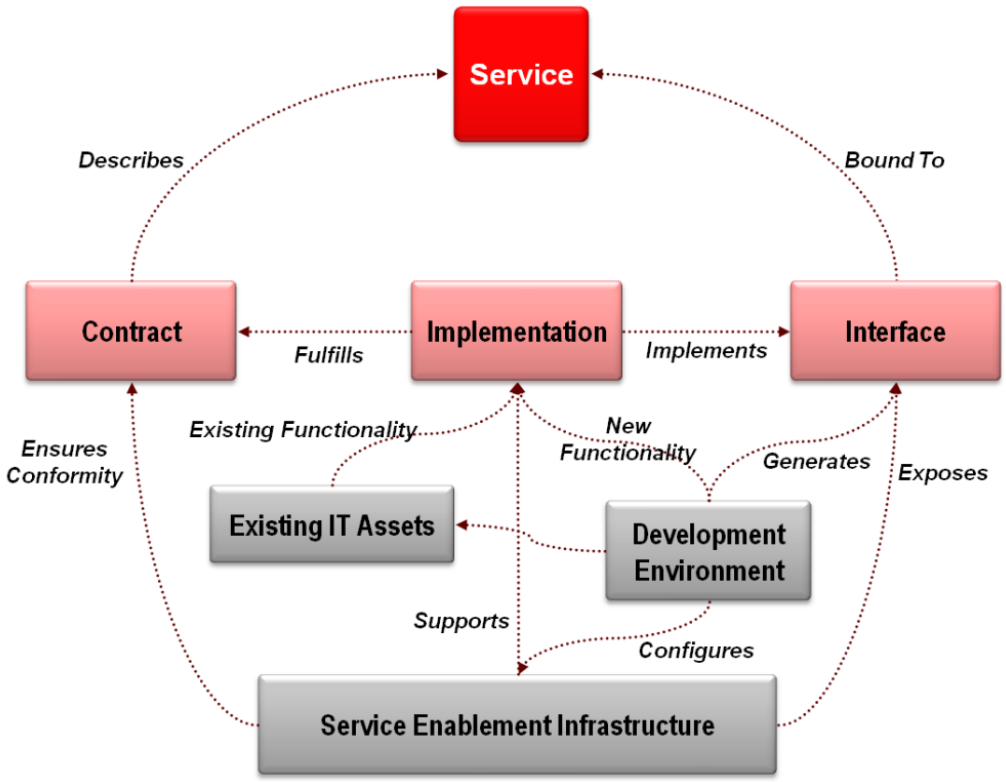
2.3 The Service Meta-Model

Following our earlier description of a Service this section expands that definition to develop Oracle's more formal models of a Service. Strictly speaking the models depicted here are "meta-models", that is to say they are models describing Services models. Ultimately meta-models can be used to describe the way we create models for specific Services. Meta-models are used here to develop a graphical representation and formalize our definition of a Service.

The OASIS definition of a Service, "a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the Service description", can be

restated as, “a Service consists of a Service description (contract), one or more capabilities (implementation), and an interface that provides access". These three facets of a Service are illustrated in Figure 2–2.

Figure 2–2 Facets of a Service



A Service is defined by its most current contract, interface, and implementation (i.e. the latest version). Multiple versions of the same Service may exist in an operational status, but this is simply to facilitate the deprecation of the old Service version. All future versions should stem from the current contract, interface, and implementation. This history with respect to the evolving contract, interface, and implementation is still maintained within the Service. When this Service is discovered, its "current" version is discovered and not any previous version.

2.3.1 Service Contract

A contract describes the Service in human-readable terms, enabling a solution designer to determine its capabilities and characteristics. It includes both functional and non-functional terms. The functional aspect of a contract describes the available operations of a Service and their functional capabilities. It should be stated using business terms, in order to promote alignment of Services to business concepts.

Contracts also specify non-functional aspects of Service, such as invocation protocols, security requirements, semantics, transaction requirements, invocation style, quality of Service, etc.

Figure 2–3 Service Contract Template

Service Contract Template

- Service Name
- Service Lifecycle Status
- Service Scope
- Service Category
- Usage Terms
- Description
- Functional Requirements
- General Information
  - Service Owner
  - Documentation availability
  - Forums, discussions, community support
  - Relevant links
  - Contacts
- Lifecycle Management
  - Current Version
  - Revision History
  - Release frequency
  - Versioning policy
  - Deprecation policy
- Interactions
  - Message exchange pattern(s)
  - Compensating Transaction Support
  - Idempotent or non-idempotent
- Quality Of Service
  - Reliability
  - Availability
  - Concurrency
  - Latency
  - Request volume
  - Maximum Message size
  - Security
- Service Enablement
  - SLA Monitoring/Enforcement
  - Exception Management
  - Management
  - Logging
- Service Policies

The Service contract specifies the purpose, functionality, constraints, and usage of a Service. The usage of the functionality embodied by a Service is governed by a contract.

A contract is made up of functional and non-functional aspects.

The functional aspect of a contract describes the behavior of available operations according to the role of the Service consumer. It is defined by the business, in business terms, which will assist in aligning IT with the business to initiate a common language.

Contracts also specify non-functional dimensions of Service usage, such as: security requirements, semantics, transaction requirements, invocation style, quality of service, etc. This content, though technical, must be provided in a human consumable format.

The primary purpose behind the contract is to provide a human readable specification for the Service which will facilitate reuse.

Architectural Principles

1. All Services must have a contract that adheres to a predefined template.

2.3.1.1 Service Lifecycle Status

The Service lifecycle tracks the Service from inception through retirement, and includes the Service's evolution through multiple Service versions. Therefore, the lifecycle can be expressed as a list of stages that should be passed.

- **Proposed** - The Service has been proposed but not yet validated.
- **Justified** - The Service has been justified and agreed upon to be realized.
- **Not Justified** - The proposed service has not been justified and will not be realized.
- **Assigned** - The service has been assigned to an owner and a development team.

- **In Progress** - The Service is currently being worked on..
- **Defined** - The Service has a defined Service contract.
- **Designed** - The Service has been designed including the creation of the interface.
- **Implemented** - The Service is realized and the implementation is documented.
- **Operational** - The Service is available for consumption in production.
- **Retired** - The Service is no longer available.

As Services are created and resued in projects, the software development lifecycle (SDLC) must be customized and co-ordinated with the Service lifecycle. As these changes are made, new Service lifecycle states will be identified.

2.3.1.2 Service Scope

The intended breadth of the consumer audience for a Service defines its scope. In general there are six primary scope boundaries:

- **Public and Multi-Enterprise** - These Services are those that will be exposed outside of the hosting enterprise and may be called by other external users such as external companies, suppliers, and customers.
- **Enterprise Wide** - These Services are available to qualified consumers throughout the enterprise. These Services are not exposed to consumers outside the enterprise
- **Intra-Line of Business (LOB)** - These Services are not available outside of the Line of Business for which they were created for. In the case of external LOB's calling this type of Service; these are acceptable on an exception basis, and are controlled by the LOB. If the number of consumers grows significantly outside of the LOB, re-scoping the Service should be considered.
- **Intra-Application** - Services that are not intended to be called outside of the scope of a given application. Typically these Services are not hosted in a shared Service environment but with the application itself. Quite often these Services result from the failed justification of a Service candidate for a wider scoped Service.

Boundaries are defined around scope because typically differing Service qualities are required for Services of differing scope. These qualities affect not only the Service contract, but the interface, implementation, and sometimes even the testing and deployment procedures.

2.3.1.3 Service Category

In order for Services to be versatile and support reuse, there must be a clear separation of concerns in terms of what they do from how they are used. Below is a list of the main service categories.

- **Presentation Services** - Performs presentation-specific formatting before information reaches the consumer.
- **Business Services (Process and Activity)** - Perform simple and/or complex units of business logic.
- **Data Services** - Accesses data from various sources and presents the data is a business friendly form.
- **Connectivity Services** - Establish connectivity with legacy applications that do not inherently provide Service-oriented interfaces
- **Utility Services** - Performs infrastructure related functions, such as security, logging, notification, etc.

Refer to the *ORA SOA Infrastructure* document for more details regarding the Service categories

2.3.1.4 Service Policies

Technical elaboration of the Service contract is performed by a software architects and analysts to derive both human and machine consumable Service policy hierarchies. Service policies are made up of potentially reusable policy fragments, possibly taking many different forms as allowed by an organization's EA standards.

Service policies are reusable, composeable descriptions of behavior used to specify Service contracts. A Service policy may be enforced directly through a facet of a Service interface (an Interface Facet), or indirectly through specified observable behavior.

Service Policies should be documented in human consumable format and should comply with enterprise, industry, or legal mandates. Policy documents should be traceable to both business requirements (ideally a Business Motivation Model) and their realization in interface and Service implementation.

Service policies may be realized in at least four distinct forms and this may be taken into account when describing policy requirements:

- Exclusively human consumable documents describing constraints on an activity, or a course of action as a response to an event. These types of policies can range from international legal mandates (e.g. constraints on carbon emissions from a manufacturing plant), thru industry/government regulations (e.g. a cap on inter-LATA telecommunications tariffs, or HIPAA restrictions on the transmission of PII), to corporate policies of an enterprise (e.g. expense reporting requirements or IT System DRS mandates).
- Rules expressed in the form of machine processable expressions: such rules may appear in the form of business process language (e.g. BPEL) "documents" or rules-engine expressions.
- Web Services specifications of policy including the base Web Service Policy Framework (W3C WS-Policy) and its various (OASIS) domain specific assertions (see Chapter 4, "Technologies and Standards" section of this document).

  Typically this level of detail is realized as part of the interface specification.
- Business rules and operational constraints can often be found hard-coded in application software during implementation: this is the least flexible (although typically highest performance) approach to realizing policy. This approach is most commonly found in previous generations of IT applications.

Policy may be arranged in connected hierarchies enabling related policies to be brought together to provide a courser grained policy, or machine processable metadata realizations of policy to be associated with human-readable documents.

The following sections elaborate further on various policy types; this list is not intended to be an exhaustive set of policy types.

2.3.1.4.1 Compliance Compliance policies can take the form of industry or enterprise standards, or regulatory and other legal mandates.

Another form of compliance document manifests in a specification from the Web Services Interoperability industry consortium (WS-I). In order to improve interoperability of Web Services, the WS-I publishes a set of "profiles". WS-I profiles correspond to core Web Services specifications (SOAP, WSDL, ...) and apply additional

requirements to restrict their use to improve interoperability. The WS-I also publishes use cases and test tools to support deployment of profile compliant Web Service.

2.3.1.4.2 Quality of Service (QoS) Statements of Quality of Service (QoS) can take many forms but they generally refer to non-functional requirements arising from specific, traceable business requirements. QoS can refer to availability (e.g. "the Service must be available 24x7"); performance requirements (e.g. "the comms system must provide dial-tone within 100ms"); transaction integrity (e.g. "debit account only upon completion of successful cash withdrawal"), etc.

Some of these statements of QoS can be realized through metadata expressions in the form of Web Services specifications. Some examples are:

- Reliable message exchange: WS-ReliableMessaging
- Transaction protection: WS-Transactions, WS-Coordination, WS-Atomic Transaction, and WS Business Activity Framework
- Message level security: WS-Security

2.3.1.4.3 Message Exchange Pattern (MEP) Basic messaging models describe a one-way flow of information, but since we need to facilitate a consumer-provider interaction, most Services operate using a request-response Message Exchange Pattern. Due to the stateless nature of asynchronous document exchange patterns (generally preferred over the synchronous, blocking, RPC style) some form of message correlation is typically needed to associate a response with its request. Correlation may be provided by the application (in which case custom identifiers are embedded within the body of a message); otherwise a Web Services standard may be used (such as WS-Addressing) to pass identifiers via a message header where they can be processed by the SOA infrastructure.

Besides the **one-way** and **request-response** Message Exchange Patterns already mentioned other examples include:

- point-to-point;
- pub-sub (publisher/subscriber);
- broadcast.

Other considerations arise from the choice of MEP including:

- Transactional requirements;
- Once only or idempotent.

Many of these considerations can be addressed in standardized ways by the SOA infrastructure rather than through costly custom application solutions. Using WSDL 1.1 each "operation" (of a "portType") can send and receive at most one message in each direction, and therefore, is only able to describe the following four message exchange patterns:

- one-way - a message is received by a Service which produces no response;
- request-response - a message is received by the Service which returns a response message;
- output-only (aka notification) - the Service sends a message but expects no response;
- solicit-response - the Service sends a message and expects a response.

The first two MEPs (referred to as inbound operations) are most commonly used while the last two (outbound operations) are not fully supported by WSDL but require the

additional support of WS-Addressing in order to bind the operations and thus determine where to send such messages. WSDL 2.0 introduces more message exchange patterns, although full description of WSDL and message exchange patterns is beyond the scope of this document.

Ideally message exchange patterns will be described by the Service contract using human readable text while the formality of WSDL and WS-Addressing, etc. will be developed during the realization of the Service interface.

2.3.1.4.4 Security Specification of security requirements overlaps somewhat with QoS, however security is such a significant concern (to most organizations) it justifies extensive handling by itself. The full subject of security is beyond the scope of this document. Please refer to ***ORA Security*** for further information on SOA security.

Beyond the specification of security requirements, SOA implementation is typically concerned with the following aspects:

- Message-level security,
- Transport-level security,
- Authentication (credential verification), authorization.

For Web Service (SOAP) Services, message-level security profiles can be represented using the WS-SecurityPolicy standard. Authentication, or access control, policies may be represented as entitlements policies. Policies for other aspects of security, such as auditing rules, may require custom-made policy definitions.

Web Service Security standards are specified by the OASIS Web Services Security Technical Committee (WSS TC) and include the following components:

- WS-Security Core Specification,
- Username Token Profile,
- X.509 Token Profile,
- SAML Token profile,
- Kerberos Token Profile,
- Rights Expression Language (REL) Token Profile,
- SOAP with Attachments (SWA) Profile.

Specification documents can be found in the OASIS document repository.

2.3.2 Interface and Messaging

A Service interface provides a means for the consumers of a Service to access its functionality according to the contract it offers. The interface separates the consumer from the Service implementation and is said to "protect" the consumer from the details of the implementation. The consumer is only able to access functions and data offered through the interface. The two parties might not even share a common programming language, as long as they are both able to interact through a common interface protocol.

Service interfaces may be reused at design time, but only a single instance of an interface can be bound to a particular Service. This simply means that two Services will not share the same runtime interface bindings, but may reuse the same physical interface specification.

In order to promote reusability, especially in heterogeneous IT environments, Service interfaces should be designed using standards-based transports and message payloads.

2.3.2.1 Transports

It is often said that SOA is not just Web Services, and conversely, using Web Services does not constitute SOA. SOA can be implemented using a number of interface protocols and messaging styles. The mechanism most commonly referred to today just happens to be SOAP over HTTP (a form of Web Services), for many reasons such as:

- Platform independence: HTTP is universally accepted as it underpins the World Wide Web. SOAP is based on XML, which is an extensible text-based language-neutral payload representation.
- Vendor support: Value-add infrastructure, such as Service management, monitoring, and mediation products tend to support this medium more than any other.
- Standards support: Enterprise quality capabilities, such as security, reliable messaging, and (to some extent) transactions have been addressed by SOAP-based standards.
- Firewall penetration: HTTP passes easily through firewalls and across wide area networks. Other protocols are often blocked by firewalls and therefore don't permeate complex environments well.

However, SOAP does have some drawbacks, such as:

- Verbosity: XML data representations are much larger than most, which involves more processing overhead and greater network bandwidth.
- Reliability and security: Not all infrastructure products support the current SOAP standards for reliable messaging, security, and transactions. Care must be taken to avoid SOAP interfaces where these qualities of service cannot be provided.

The following principles pertain to Service transports:

- Transports conforming to SOAP protocol should follow WS-I.

2.3.2.2 Payloads

Service payloads may be designed in a number of ways. Assuming a SOAP interface, the payload options may include basic typed parameters, such as strings and integers, or document style input arguments and return values. SOAP may include attachments, which is handy for passing objects such as images or other binary content that doesn't need to be parsed by the Service or intermediaries.

The following principles pertain to Service payloads:

- All Services must use document style parameters in the form of one document as an input and one document (or void) as a return.
- Input documents must include a client identifier as an attribute of the primary element.

2.3.3 Implementation

The implementation is the technical realization of the contract. It is responsible for fulfilling all functional and nonfunctional capabilities stated therein. The implementation may leverage functionality in existing systems, newly developed code, or some combination of both.

Since infrastructure is often used to help satisfy certain capabilities of the Service (either functional or non-functional), the infrastructure components act as part of the Service implementation. Therefore, an implementation will often consist of deployable code as well as infrastructure configurations, security policies, management agents, etc. - all parts working together to fulfill the contract.

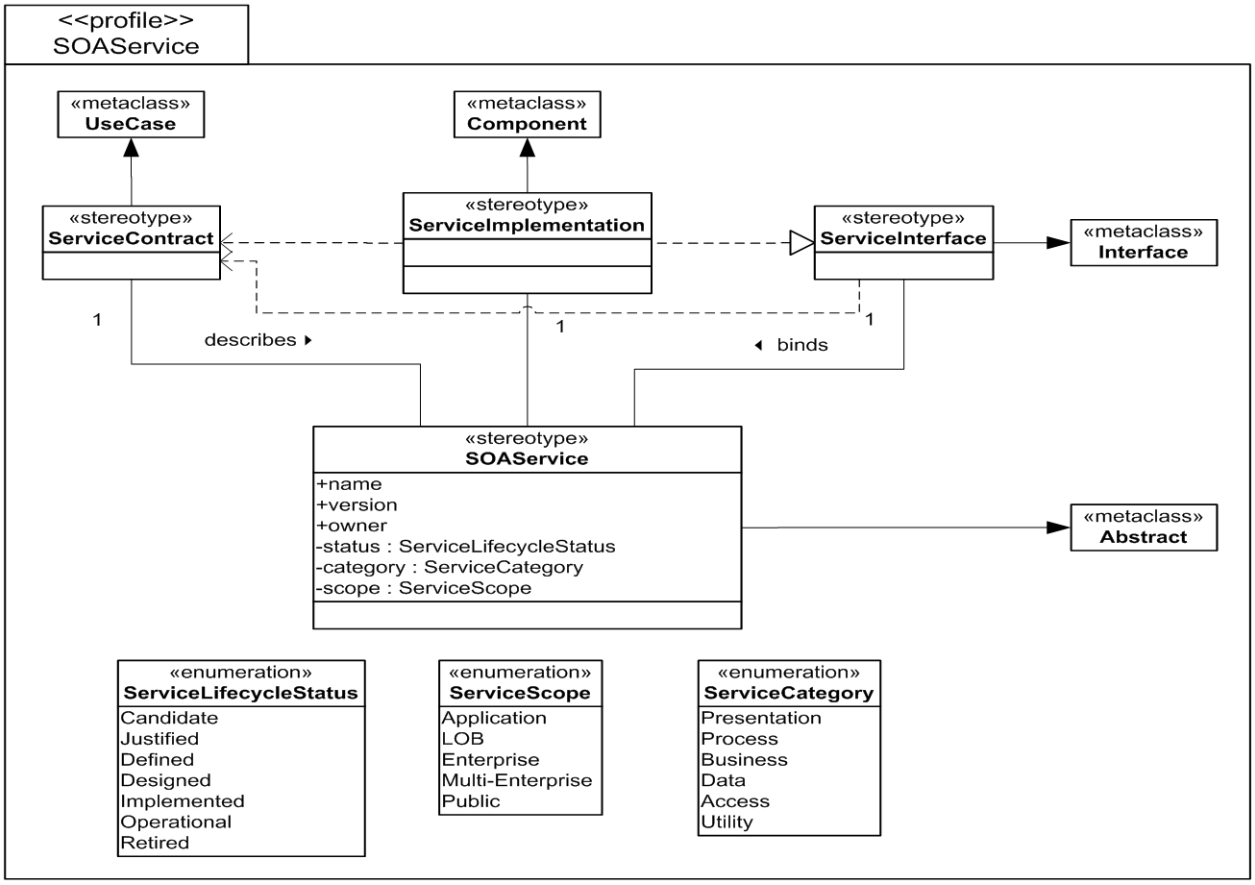
The Service infrastructure side typically provides the Service enablement capabilities for the implementation. These capabilities may include: exposing the interface as a Web Service, handling SLA enforcement, security, data formatting, and others. Service infrastructure should be utilized when possible, as it reduces the burden on Service providers, from an implementation standpoint.

2.4 A UML Profile for Services

The Service meta-model shown in Figure 2–2 provides a visual depiction of the aspects of a Service. It does not, however, provide the rigor of a UML profile that would be suitable for modeling Services via UML. The remainder of this section highlights aspects of the ORA UML Profile based on the components of a Service that were defined and described earlier in this chapter.

The diagram in Figure 2–4 below outlines the complete set of extended UML classifiers introduced by the ORA UML profile for Services. This Services profile package identifies four new stereotypes and their underlying UML meta-class elements.

Figure 2–4 UML Profile for a Service



The diagram above uses UML2 standard notation to denote the extension of existing, standard UML meta-classes for the creation of a specialized instance of a classifier. In the case of the Service UML Profile (above) four new stereotypes are introduced as extensions to underlying UML meta-classes, these are:

- SOAService: extending the Abstract classifier, associating the Service Contract, Service Interface, and Service Implementation, and having it's own properties: name, version, owner, status, category, and scope.
- ServiceContract: extending a Use Case (or collection of use cases) and corresponding use case specification, describing a coherent unit of functionality of a SOA system. The ServiceContract stereotype should be used in the case of either producer or consumer contracts.
- ServiceInterface: extending the UML Interface, specifying operations of the Service described by the ServiceContract
- ServiceImplementation: extending a UML Component, realizing the operations of the ServiceInterface while conforming to constraints and implementing the functional requirements of the ServiceContract

This UML extension package defines a Service as an abstract container relating the three principle facets of a Service: the Service contract, the Service implementation, and the Service interface. The abstract Service container carries little information (or operation) of its own but instead ties together one each of the contract, interface, and implementation with a single version identifier (i.e. an instance of this Service will be a "Service version"), lifecycle, type, and scope.

In practice any classifier can be used to describe the Service and its three key facets, contract, interface, and implementation; however, the underlying UML meta-classes chosen here serve to highlight the separation of their roles:

- The SOAService itself is abstract because it has no real manifestation in an implementation, but serves as a convenient concept during analysis when it may be used in business analysis, early Service identification, high-level collaboration / orchestration, lifecycle analysis, etc.
- The ServiceImplementation is best served using component diagram elements since this notation typically arises in design, which is the phase in which we start to consider implementation details.
- The ServiceContract is represented here based on the UML use case since it is a specification of behavior of the Service, its functional and non-functional requirements, and constraints. The ServiceContract describes a use case of the (SOA) system.
- The ServiceInterface is naturally based on the underlying UML interface meta-class.

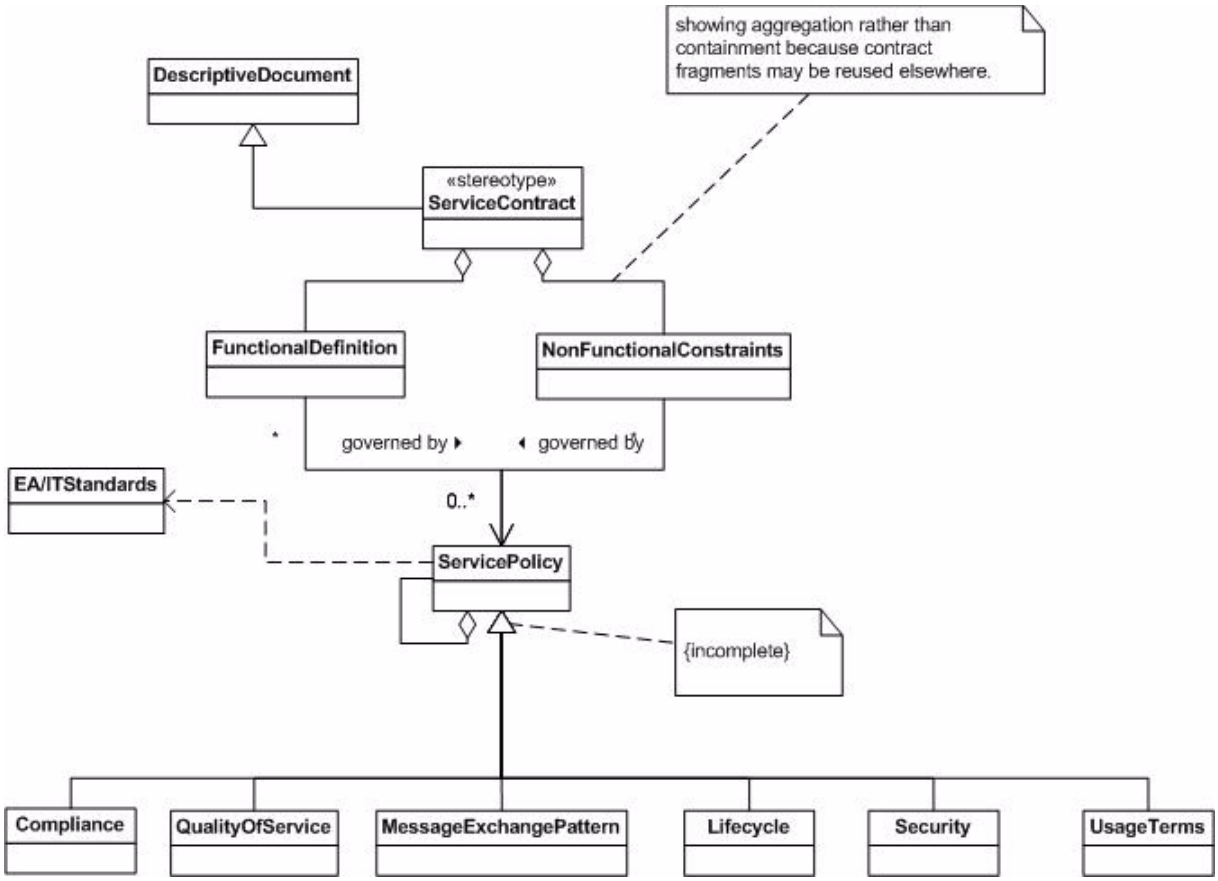
The Service itself is further supported by enumerations for the properties status, category, and scope. The enumerations of the properties are dependent on the SOA reference architecture and Service engineering framework being used.

The ServiceLifecycleStatus is special case since it is related to the software engineering lifecycle and, as such, it is more of a project management concern than a direct attribute of the contract. For this reason it is likely to appear in UML diagrams using tag property notation rather than classifier attributes. The choice of lifecycle states may vary depending upon a customer's engineering and configuration management practices.

2.4.1 Service Contract Meta Model

The figure below shows the meta model details for the service contract stereotype. Here we can see the contract is made up of functional and non-functional descriptions which use policy fragments.

Figure 2–5 Service Contract Meta Model



The Service Contract stereotype meta model extends the description of the ServiceContract from the SOAService UML profile defining the following key statements:

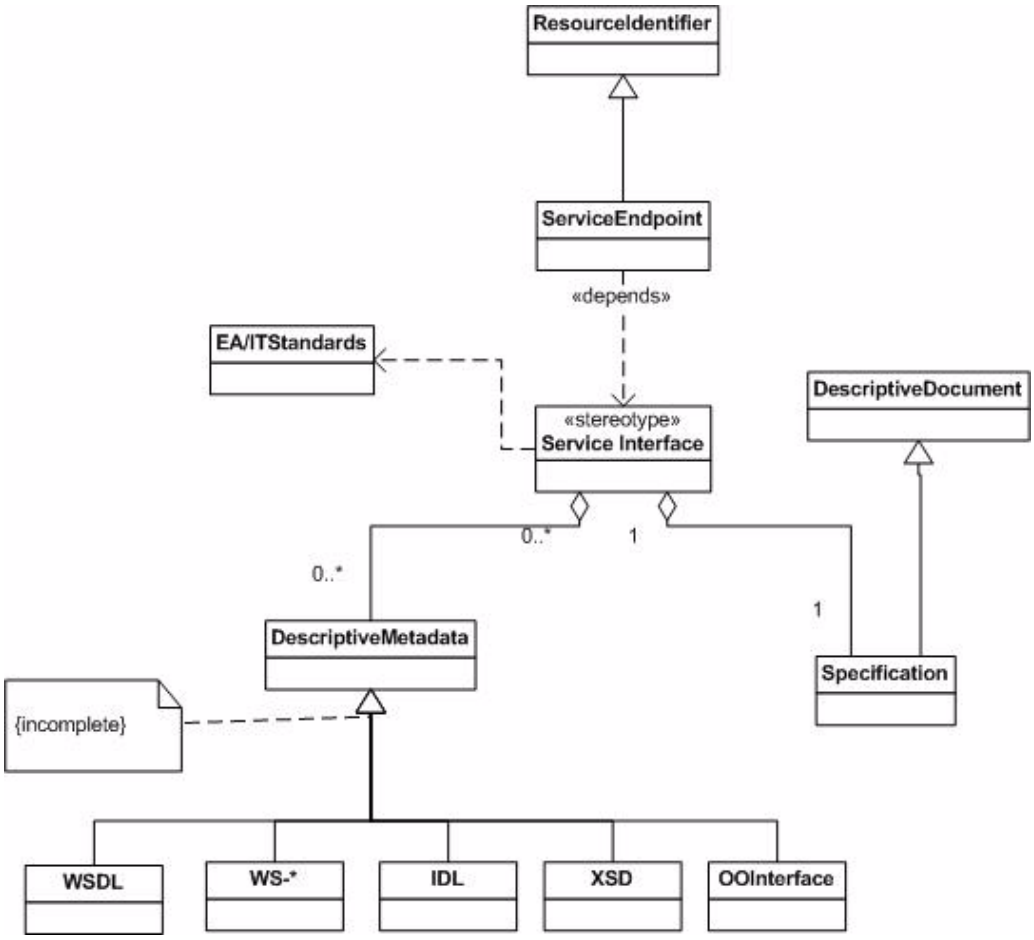
- The ServiceContract is a DescriptiveDocument.
- The ServiceContract documents (aggregates) functional and non-functional requirements (descriptions and constraints) for the Service.
- The functional and non-functional requirements, while typically descriptive and/or structured documents, may also be defined by policies.
- ServicePolicy(s) may be reusable and arranged hierarchically for any given Service contract. Examples of policy types are shown in the remaining classifiers.
- ServicePolicy(s) must conform to EA/IT standards.

2.4.2 Service Interface Meta Model

The figure below shows the meta model details for the Service interface stereotype. Here we can see the interface is made up of (aggregates) a service specification

(another descriptive document) and descriptive metadata, highlighting the human and machine readable aspects of the interface definition.

Figure 2–6 Service Interface Meta Model



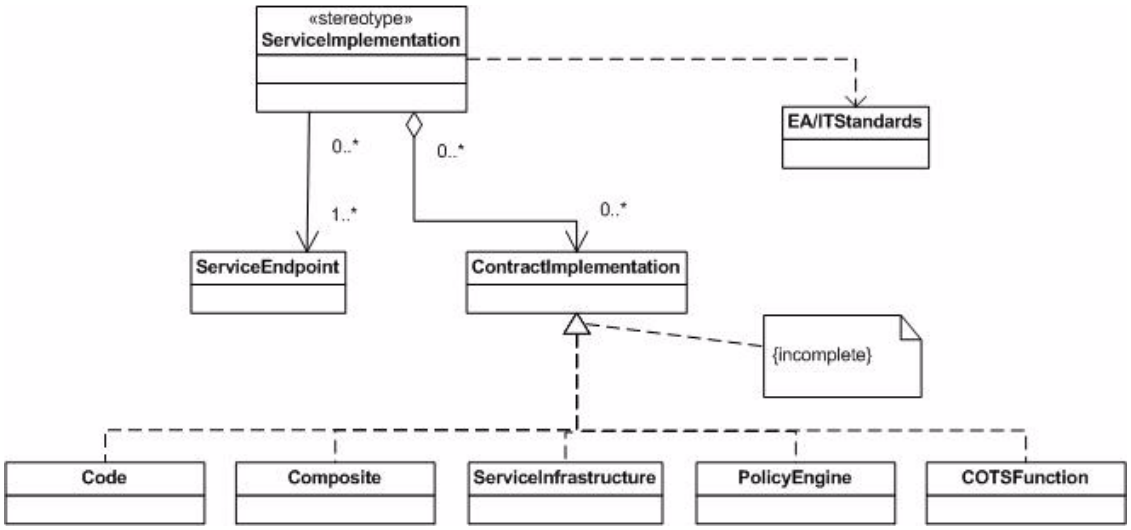
The Service Interface stereotype meta model extends the description of the ServiceInterface from the Service Profile making the following key statements:

- The ServiceInterface aggregates (reusable) various types of (implicitly machine readable) descriptive meta-data and (implicitly human readable) DescriptiveDocument specifications.
- A runtime instance of the interface is uniquely defined by its endpoint implementation while facets of the ServiceInterface are otherwise reusable.
- The ServiceInterface depends upon the ServiceEndpoint implementation and must conform to IT/EA standards. The ServiceEndpoint is a resource identifier (e.g. URL).

2.4.3 Service Implementation Meta Model

The following figure shows the meta model details for the service implementation stereotype. Here we can see the implementation is made up of (aggregates) various manifestations of contract implementation.

Figure 2–7 Service Implementation Meta Model



The Service Implementation stereotype meta model extends the description of the ServiceImplementation from the Service Profile making the following key statements:

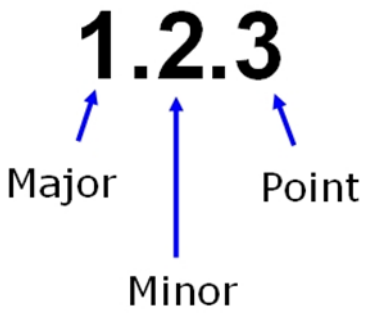
- The ServiceImplementation aggregates ContractImplementation elements.
- The ServiceImplementation is bound to one or more end points (linking it to the interface specification).
- The ServiceImplementation must conform to IT/EA standards.

2.5 Service Versioning

Whether it is from proactive activities such as new or changing business requirements or reactive activities such as optimizations and corrective actions, Services are bound to change and evolve over time. Changes can be accommodated in a variety of ways, however, following the loose coupling principles of SOA, the impact to consumers should be minimized. For this reason changes should be handled as versions, ideally supporting the ability to have multiple versions running concurrently. There will be times when this is not feasible, but care should be taken to minimize these instances. Failure to do so requires consumers to deploy updates in lock-step with Service updates, which can quickly become unwieldy as usage increases.

A number of architectural principles and guidelines supported by an appropriate Service versioning strategy is required to address the eventuality of having to version a Service. The lack of a Service versioning strategy and implementation greatly impacts the flexibility of an SOA as it severely hinders Service evolution. The versioning strategy adopted does not need to be overly complicated, but from a best practices perspective, the Service version should at a minimum consist of a numeric 3-tuple (major, minor, and point).

Figure 2–8 Versioning Tuple



The appropriate element of the tuple is incremented based on the type of change that will be applied to the Service.

- **Major** - A major change to the Service would result in the first element of the tuple to be incremented. These types of changes are not backwards compatible and will therefore require Service consumers to be modified before being able to utilize the new version of the Service. If Services tend to go through a large number of major revisions, it is an indication that there are areas for improvement, particularly around the area of Service identification and discovery. The deployment of a major release of a Service typically requires the previous version to be co-deployed. The previous version is considered to be deprecated in order to allow Service consumers to be upgraded to the newest version of the Service.

  A major change to a Service can:

  - Allow deprecated operations to be dropped,
  - Allow a change in an operation signature in a manner that could break consumer compatibility,
  - Replace input and output message type with a different input and output message type,
  - Be used if Service has gone through a major rewrite or has been updated considerably.

- **Minor** - A minor change to the Service would result in the second element of the tuple being incremented. These types of changes are backwards compatible, and do not require dependant Service consumers to be updated in order to utilize the new Service version. The deployment of a minor release to a Service results in the immediate retirement of the previous version of the Service.

  A minor change to a Service version can:

  - Allow operations, data types, optional fields to be added (but not deleted),
  - Must not break any consumer compatibility.

- **Point** - A point release of a Service represents the least significant types of changes for a Service. Changes such as defect fixes, where no new functionality is added to the Service, are classified as point releases. Like the minor release, the point release is backwards compatible and upon deployment the previous version of the Service is immediately retired.

  A point change to a Service version:

  - Can be used when fixing specific problem (e.g. bug, performance),
  - Does not allow new features to be added.

2.5.1 Versioning the Components of a Service

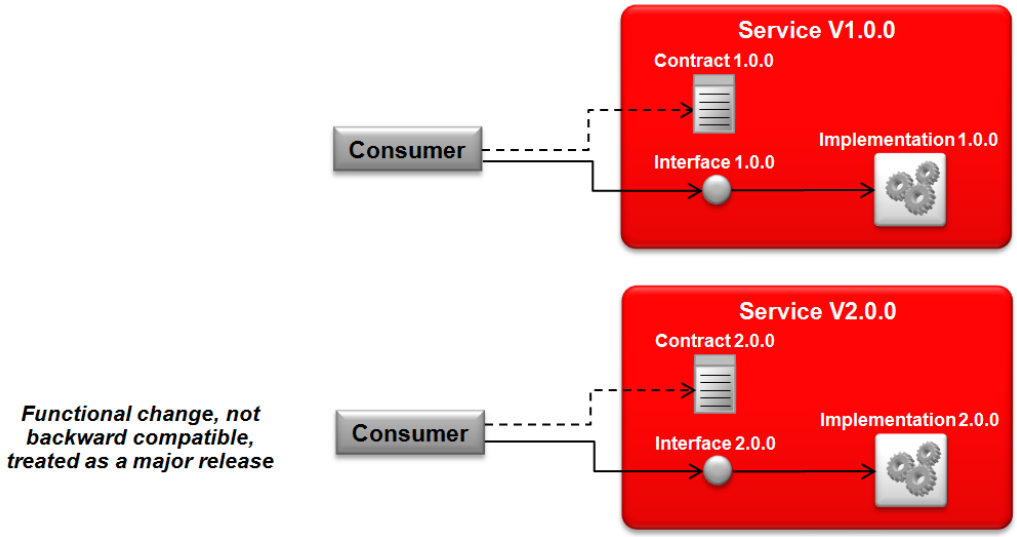
The components of a Service (contract, interface, and implementation) may evolve independently. But the revision tuple must be synchronized across the components and with the resulting Service itself.

As best practice on each major release, all components of the Service are incremented to the same tuple with the same release number and minor and point numbers set to 0. Each change to any of the components will increase the corresponding tuple number of the Service. The component changed will then inherit the tuple of the Service. This ensures consistency in the increment of the Service version number while at the same time the versions of the components will indicate the scope of change applied to the version.

2.5.1.1 Contract Versioning

Service requirements additions and changes drive contract versioning. Functional changes will most likely involve interface and/or implementation changes.

Figure 2–9 Contract Versioning - Major Release



In Figure 2–9 as the functional change is not backward compatible, it is treated as a major release. Therefore, the version number of all components (contract, interface, and implementation) have been incremented to align with the Service version number.

Figure 2–10 Contract Versioning - Point Release

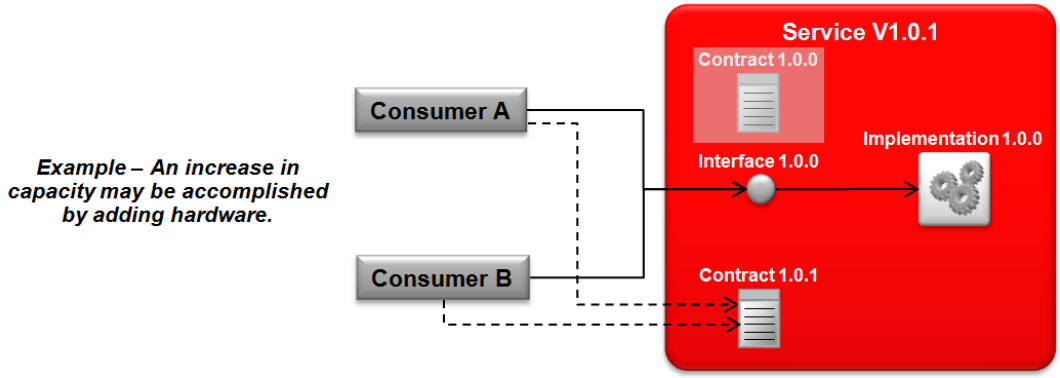


Figure 2–10 highlights that a non-functional contract change that does not affect code can be treated as a point release. Therefore only the contract version has been incremented to align with the Service version number.

2.5.1.2 Interface Versioning

As highlighted earlier interface changes are required for incompatible parameter data type changes, the additional/removal of operations, and changes in operation semantics.

Figure 2–11 Interface Versioning

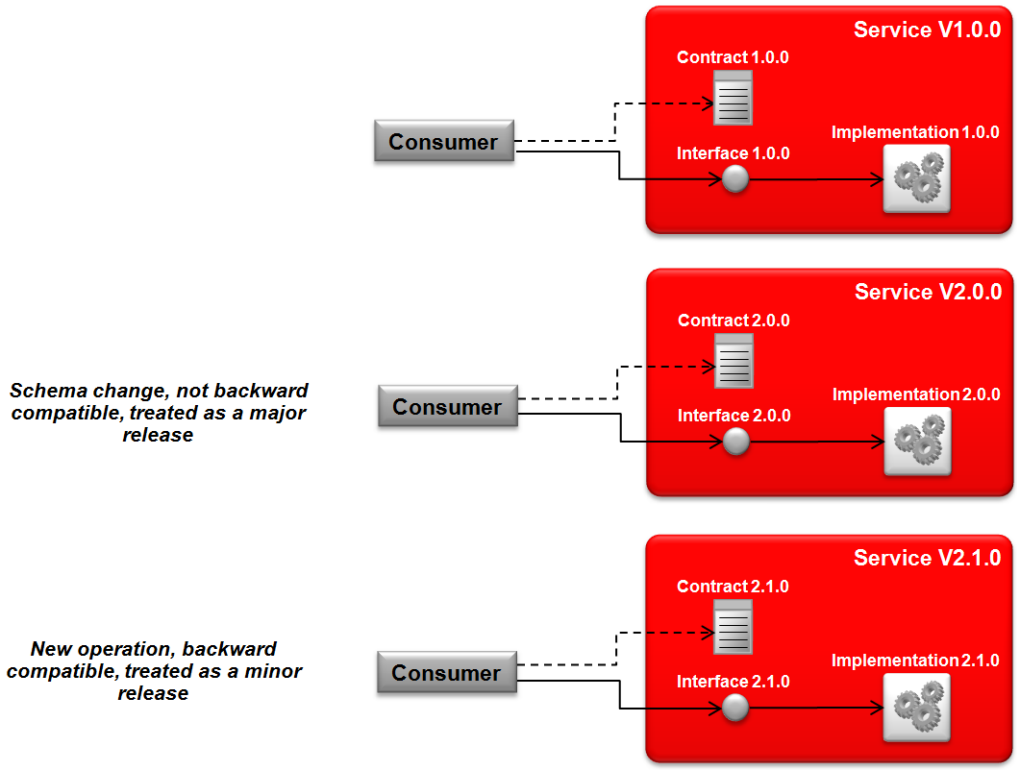


Figure 2–11 highlights multiple versions of a Service deployed where different consumers are consuming different versions of a service.

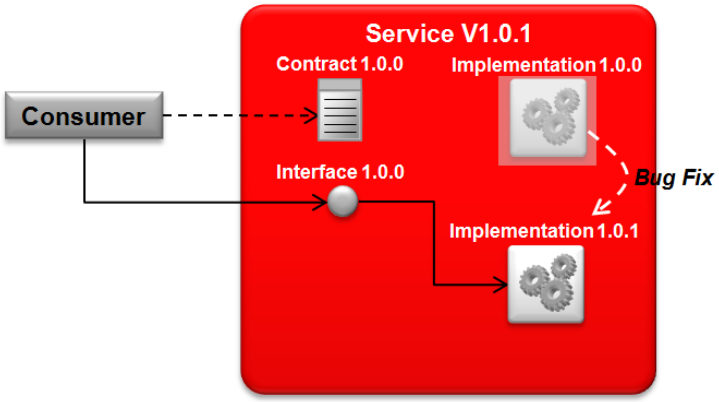
When there is a schema change that is not backward compatible, it is treated as a major release. Therefore, the version number of all components (contract, interface, and implementation) have been incremented to align with the Service version number. (i.e. V2.0.0)

When there is a new operation that is backward compatible, it is treated as a minor release, and all of the Service components are incremented to align with the Service version number. (i.e. V2.1.0)

2.5.1.3 Implementation Versioning

Changes in the implementation without interface or contract changes, should be transparent to the users (i.e. no re-coding necessary).

Figure 2–12 Implementation Versioning



Architectural Principles

1. Services must support multiple concurrent versions.
2. Service providers must be able to release new versions into production without waiting for consumers to certify on them.
3. Consumers must be able to test and certify on new Service versions before switching to the new version in production.

Additional Guidelines

1. Service providers should base their testing on the contract (black box testing by contract). Consumers may certify for their use based on their own functional test criteria.
2. Service consumers should not require code modifications in order to access the latest compatible Service versions (point releases).
3. Consumers should determine which version they access.

Regarding this last guideline, some SOA environments may alternatively decide that Service infrastructure should determine which versions they access; in any case, one approach should be selected and used consistently.

3

Conceptual View

This chapter presents a conceptual view of the Service-Oriented Architecture. It describes the core architecture capabilities, key principles, and a conceptual model that is unencumbered by implementation details.

3.1 SOA Capabilities

An effective deployment and adoption of SOA will require some key infrastructure capabilities. The infrastructure capabilities must be aligned with the business goals to enable quicker delivery of business capabilities. This will enable the business to achieve faster time-to-market benefits of SOA. A prudent SOA investment will require that the infrastructure capabilities are built in a prioritized order. Growing the infrastructure capabilities is a prerequisite to improving SOA maturity as discussed in a later section

Figure 3–1 organizes the key SOA capabilities into five primary domains - Core, Management, Governance, Security, and Monitoring.

Figure 3–1 SOA Capabilities

Some SOA capabilities are dependent on basic infrastructure capabilities such as messaging and security. There are also enterprise technological strategies such as BPM and EDA that complement SOA. The capabilities of these technological infrastructures can leverage and build upon the SOA infrastructure capabilities. This document does not cover these in detail, but they will be addressed in the appropriate technology perspective documents. A brief overview of these capabilities is listed in Section 3.1.6.

3.1.1 Core

This section describes the core SOA infrastructure capabilities essential for building an enterprise-class SOA. The core SOA capabilities include mediation, message transformation, service routing, dynamic binding, error handling, and policy enforcement

3.1.1.1 Mediation

Mediation can be broadly defined as resolving the differences between two or more systems in order to integrate them seamlessly. A typical IT architecture has a variety of systems and components that are fundamentally different. A better alternative to embedding the mediation logic into each of these systems would be to provide the mediation capability in the SOA infrastructure.

A Service-Oriented Architecture can exist with Services exposed using alternate means in addition to Web Services. In order to be flexible and allow for a wide variety of heterogeneous service invocation techniques, SOA will need the support for transport mediation, multiple message formats, over various invocation strategies. Transport mediation bridges the differences in the communication protocols like HTTP, File, and FTP.

- **Message Exchange Pattern (MEP) Mediation** - MEP mediation resolves the difference between the invocation patterns of the consumer and provider. The most common MEP used in SOA is the synchronous request/response pattern. Asynchronous one-way, fire-an- forget, asynchronous request/response, and robust one-way are some of the other popular MEPs that should be supported by the SOA infrastructure. A good SOA infrastructure should also be able to mediate different MEPs using sync-to-async or async-to-sync bridging.
- **Transport Mediation** - Transport mediation allows the consumer to invoke a Service using a different transport than that supported by the provider. The SOA infrastructure achieves it by translating the transport protocol and transport headers. The set of supported transport protocols is important to consider when selecting SOA infrastructure for composition. Some of the common routing transports that may typically be used to invoke services or pass messages within an SOA are HTTP(S), File, FTP, JMS, RMI, IIOP, **JCA**, and POP/SMTP/IMAP.

  Multiple transport protocols may be used in order to address various Quality of Service (QoS) requirements. While SOAP over HTTP(S) is the most commonly mentioned form of transport due to its ubiquitous nature, it does not often support reliable messaging and transactions. Even though WS-ReliableMessaging and WS-Transactions specifications have been written, not all entities support them. Therefore, it is quite common to leverage a transport that does, when such qualities are needed
- **Security Mediation** - Just as not all service providers support the same technologies and transports, they do not all support the same security implementations. Therefore, in order to be able to mediate between endpoints, the infrastructure must be able to mediate issues of security. That is, it must be able to meet the security requirements of a service by extracting and converting security information provided to it by the service consumer. In this way it is acting as a converter rather than a negotiator between endpoints. More details on the security mediation can be found in the *ORA Security* document.

3.1.1.2 Message Transformation

Transformation is a type of mediation that is a critical capability of any SOA infrastructure. The ability to manipulate and transform messages as they travel from consumer to producer, and optionally back to the consumer, provides the designer with a great deal of flexibility. These capabilities provide support for such concepts as transformations to and from canonical message formats. They also allow for Services to be utilized for a wider audience, by being able to adapt the input and output to a Service invocation. Message transformation might include aggregation, enrichment, filtering, and wrapping.

3.1.1.3 Service Routing

Routing refers to the ability to control where a message is sent or which Service endpoint is invoked. This can be accomplished by extracting data that affects routing decisions from the contents of the request message (content-based routing), message header, or via configuration data (configuration-based routing).

Loose coupling is the ability to decouple the Service consumer from the implementation specifics of the Service provider. With loose coupling a Service provider can be versioned, physically moved, or replaced without affecting the Service consumer. Routing capability enables the loose coupling required for SOA. These features are fundamental to managing and maintaining a mature SOA deployment.

3.1.1.4 Dynamic Binding

Dynamic binding refers to the act of connecting a Service consumer to a producer at runtime. The SOA infrastructure must determine which Service producer to use and return endpoint information to the consumer, who then accesses the appropriate endpoint.

Dynamic binding satisfies the need for loose coupling; however, it requires two round trip interactions, at least the first time - one between the consumer and SOA infrastructure, and the other between the consumer and producer.

3.1.1.5 Error Handling

Error handling capabilities allow expected and unexpected error conditions to be planned for and handled appropriately at design time. Error handlers are placed within a particular scope which allows errors to be isolated and handled at the most appropriate point; thereby providing the highest likelihood of successful recovery from the error.

3.1.1.6 Policy Enforcement

An important capability of the SOA infrastructure is the ability to define and enforce policies. By nature, SOA infrastructure is loosely coupled and distributed. Policy enforcement provides a way to define common policies independent of the Service implementation and apply them on an as needed basis. Policy enforcement can happen at different enforcement points of the infrastructure based on the use case. Policies can also change independent of the Services or applications allowing a higher degree of agility and control. Policies may include regulatory compliance rules, business policies, security policies, SLA policies, and validation rules. The SOA infrastructure should be able to support the enforcement of a variety of policy types.

3.1.2 Governance

An effective SOA Governance will require a minimum of the following capabilities.

- Asset Management
- Portfolio Management
- Asset Lifecycle Management
- Asset Version Management
- Usage Tracking
- Service Discovery
- Policy Management
- Dependency Analysis

Each of these SOA governance related capabilities is described below:

3.1.2.1 Asset Management

Asset management is a key governance capability of a SOA infrastructure. SOA requires the management of the metadata of a variety of assets like Service artifacts, contracts, policies, and schemas. A robust asset management infrastructure that can maintain the metadata of these assets and relationships between them is a critical aspect of effective SOA governance.

3.1.2.2 Service Portfolio Management

An important point to recognize with SOA is that the creation and lifecycle maintenance of Services adds some amount of overhead to the development process. The amount of overhead varies, depending on the formality, complexity, and rigor involved in identifying, discovering, creating, managing, and governing Services. Even if steps are taken to greatly reduce overhead, one can expect some amount to exist.

Given this acknowledgement of overhead, Services should be planned and managed to best satisfy the needs of business and IT. This requires the ability to know what Services exist, what capabilities they provide, and ideally, what business processes they pertain to. The Service portfolio is a way to represent this knowledge, communicate it, and plan for future Services.

3.1.2.3 Asset Lifecycle Management

A corollary to portfolio management is lifecycle management. Services have a lifecycle, starting with project requirements, continuing with identification and creation of a Service, deployment, and ending with eventual retirement. Lifecycle management involves the strategy and planning around stages of the Service lifecycle. It supports governance in terms of what Services are approved and the status they are given. It also supports release planning, as lifecycle management includes control over Service releases on which applications and other Services may depend.

3.1.2.4 Asset Version Management

A natural extension of lifecycle management is version management. Services may require changes based on a number of factors, such as code defects, functional changes, non-functional changes, resource changes, etc.

Versioning can be fairly simple for Services with only one consumer. In some cases the consumer and provider may be updated simultaneously. This reduces (or eliminates) the need for concurrent versions.

Versioning becomes more difficult when Services are used by multiple consumers. In these cases, every effort should be taken to support multiple concurrent versions in production. This allows each consumer to test and migrate to the newer version under their own release cycles.

3.1.2.5 Usage Tracking

Usage tracking comes in three forms:

- the registered intent to use a Service
- the interest in a Service or Service asset resulting in a subscription to that asset
- the actual audit trail of consumers invoking Services.

The first two are design-time tracking mechanisms, while the third is a run-time approach. All are important in their own way.

There are certain benefits that can be realized through the use of **Usage Agreement**s, aka **Consumer Contract**s. These documents formalize the intent to use a Service, which removes ambiguity in terms of who is relying on a Service and what performance, load, and availability requirements exist. Consumers may feel more comfortable under such an agreement because their needs are well known. Providers can use this information for lifecycle management and capacity planning. These benefits make it a recommended way to track and manage Service usage.

3.1.2.6 Service Discovery

For all practical purposes, Service discovery is a design time exercise. The primary reasons involve semantics and integration.

Semantics specify exactly what a Service does, how it behaves in normal and exception cases, and the meaning of each data element sent and received. This is something that requires thought and reasoning. The SOA infrastructure helps in this regard by providing a place to manage all the information required to make semantic-related decisions.

Integration is simply the act of coding or configuring the interactions involved with Service invocation. It includes the steps necessary to prepare for Service invocation, as well as the steps to process normal and exception-case results. While tooling can help accomplish some of this through configuration, it is difficult to imagine the entire integration scenario taking place dynamically at runtime. Note that runtime integration is different than runtime binding. Conceptually, runtime binding is the act of connecting a consumer with a provider. The consumer and provider must already be compatible, both in terms of semantics and integration. The connection can be made either through routing decisions or templates (using a Service registry). But in either case, the Service discovery aspect must be performed first in order to overcome semantic and integration hurdles.

3.1.2.7 Policy Management

SOA enables policy driven architecture that decouples specific business rules, regulatory compliance checks and security policies to be modeled and deployed independent of the Services. This abstraction gives the flexibility to model once and deploy multiple times in addition to allowing centralized control over these business policies. Policies allow the dynamics of the system to be quickly changed and provide visibility into the business by plugging into the SOA management frameworks. SOA infrastructure should include capabilities to manage, distribute, and monitor the policies.

3.1.2.8 Dependency Analysis

As the maturity of SOA increases, the number of Services and other interdependent assets will grow rapidly as well. These interdependencies are hard to track if not managed properly. An important part of SOA governance is the ability to assess the impact in development and operational environments as a result of a change to a given asset. Dependency analysis is beneficial as the number of Services increases, the relationships become more complex, and the need to revise or retire Services arises. It makes it easier to perform impact analysis when changes need to occur. This goes both ways - one may need to understand the dependencies a Service has, if the Service needs to be changed, moved, or virtualized; or one may need to understand what Services depend on a particular resource or Service, if the resource or Service needs to be modified, retired, or moved

3.1.3 Management

Service management refers to the configuration of SOA Infrastructure to control the runtime aspects of the deployment. The following capabilities are commonly looked for when evaluating Service management infrastructure:

- Service Level Agreements (SLA) Management
- Logging and Monitoring
- Versioning Support

- Resource Browsing
- Environment Propagation

These capabilities are discussed in detail below.

3.1.3.1 Service Level Agreements (SLA) Management

The ability to configure SLAs on Service end points and infrastructure-provided Services on the following attributes is always beneficial:

- Average processing time of a Service
- Processing volume
- Number of errors, security violations, and schema validation errors

The ability to configure alerts for SLA rule violations as well as enable or disable Services based on this data is also very useful.

3.1.3.2 Logging

The ability to configure the level of logging and auditing is an important capability of the Service management infrastructure.

3.1.3.3 Versioning Support

The infrastructure should provide the ability to stage the deployment of multiple versions of a Service which will allow multiple versions of a Service to be available at runtime. The ability to provide routing and transformation between versions is an essential infrastructure feature that ensures that the right version of the Service is consumed. Different consumers might want to access different versions of the Service for various reasons, and the infrastructure should route the requests appropriately.

3.1.3.4 Resource Browsing

SOA infrastructure is made up of several moving parts, such as Service endpoint information, schemas, transformations, WSDLs, and policies. A certain level of automation is required to ensure that the resources are detected and auto-configured. All resources should be kept in sync and should act in unison to run the Service successfully. That makes it important to be able to browse the resources and their configurations exposed through the SOA infrastructure so that they can easily be registered and managed.

3.1.3.5 Environment Propagation

Since SOA infrastructure is heavily configuration based, the ability to propagate the configuration information from environment to environment is very important. This should include the ability to override system specific settings that vary from environment to environment (such as host names, etc.).

3.1.4 Monitoring

The monitoring related SOA capabilities include:

- Runtime Service Usage Tracking
- Exception Management
- Performance Management
- SOA Dashboard

These capabilities are discussed below.

3.1.4.1 Runtime Service Usage Tracking

In complicated SOAs with potentially hundreds (or even thousands) of participants, it becomes increasingly important to have a centralized point of tracking and analyzing data related to the operation of both SOA infrastructure and the Services participating in the SOA. The SOA monitoring infrastructure analyzes, stores, and acts upon runtime data to ensure the optimal operation of the runtime environment. It also provides the information back to the operations team which enables informed decisions to be made about scaling the infrastructure or whether there is room to expand the usage of the infrastructure and its Services across additional applications and Service consumers. Service usage tracking gives the ability to track what Services are being accessed and by which consumers, and version usage tracking gives the ability to track which versions of a Service are being used and by whom.

3.1.4.2 Exception Management

The SOA infrastructure should be capable of monitoring and analyzing the exceptions that occur at various parts of the infrastructure. There are several types of exceptions that need to be handled by the infrastructure, including:

- Functional exceptions
- Business exceptions
- Service availability exceptions (show how often the Service is available or unavailable)
- Security violations (show the attempts to use a Service without proper access rights)

3.1.4.3 Performance Management

Response-time data for each Service and the system load, optionally plotted over time of day and days of the week, would be vital information for managing the performance of the system. The infrastructure should aggregate and provide the performance metrics to assist troubleshooting and process improvement activities. The system performance should be closely monitored and proactively acted upon to ensure that the Services are highly available and meet the SLA requirements.

3.1.4.4 SOA Dashboard

A SOA dashboard is a system that collects data from several Service based systems, aggregates them, and measures the results against metrics in order to provide an interpretation in the form of control panels and reports. By defining a dashboard based on metrics around SOA, the system will provide the management with the means to monitor the SOA adoption process and identify strengths and weaknesses in the process and organizational areas to act upon accordingly.

3.1.5 Security

Due to its inherent distributed nature, SOA can greatly complicate the security landscape of an IT organization. It stands in contrast to silo'ed applications, which are typically secured by adding layers of protection around the perimeter, in favor of distributed functions and data, which are much more open, and potentially vulnerable. The SOA security infrastructure is meant to address this problem. It extends from the security infrastructure already in place to meet the challenges presented by the adoption of SOA.

The security infrastructure must address the ability to secure messages, e.g., message level security, as well as the ability to ensure that functions and data are accessible to the correct audience and under the right conditions. It must do so in a way that is scalable and yet manageable. This involves the unification of assets that drive security decisions, such as LDAP directories, databases, etc., as well as the centralization of policy management.

The security infrastructure provides the standard capabilities, such as authentication, authorization, encryption, credential mapping, non-repudiation, confidentiality etc., to the other SOA components in addition to the SOA specific capabilities. These capabilities may be offered as Services, much like other types of Services, that can be discovered, versioned, and invoked through standards-based interfaces. Or, they may be accessed by application containers through low level APIs.

SOA security is covered in more detail in the *ORA Security* document. The SOA specific capabilities of the security infrastructure are summarized below.

3.1.5.1 Standards-Based Security

The SOA security infrastructure must enable choice and interoperability through the support of industry accepted security standards. A number of security standards such as WS-Security, WS-SecurityPolicy, XML signatures, XML encryption, and SAML, enable security interoperability between the disparate components of the SOA infrastructure. Standards also ensure that the security technologies are compatible with existing security products and capable of being leveraged across a diverse array of web servers, application servers, and custom applications built in various languages.

3.1.5.2 Security Policy Provisioning

The security infrastructure must efficiently distribute incremental updates to policy and configuration data and ensure synchronization across the enterprise. The infrastructure must allow provisioning of security policies that control access and authorization.

3.1.5.3 Distributed Policy Decision-making and Enforcement

The SOA security infrastructure must provide a means for policy decisions to be defined centrally and enforced locally to meet performance requirements. Some of the security decisions can be pushed to the periphery through an agent-based architecture to improve the performance of the transaction and integrity the system. Distributed policy decision points and policy enforcement points allow a dynamic and flexible architecture that can respond to the needs of the business much faster.

3.1.5.4 Security Management

SOA infrastructure should provide the following security management capabilities:

- The ability to manage policies for authentication, encryption and decryption, and digital certificates as defined in the Web Services Security (WS-Security) specification.
- The ability to enable traditional transport-level security for HTTP and JMS protocols by utilizing SSL.
- Support for one-way and two-way certificate based authentication, as well as providing support for HTTP basic authentication is needed to support basic security requirements.

3.1.5.5 Centralized Security Management

The security infrastructure should provide an integrated enterprise policy "system of record" that eliminates fragmentation across disparate applications. The security management component of the SOA infrastructure allows configuration of the policies around SOA resources centrally at design-time, and then publishes the policies to the relevant security infrastructure responsible for enforcement at runtime.

3.1.6 Other Complementary Capabilities

SOA is an architectural style that can be applied to various technology strategies. A comprehensive IT infrastructure will require much more than the SOA capabilities. A number of technology strategies like BPM, EDA, BI, and Enterprise 2.0 can take advantage of SOA and provide complementing capabilities that help complete the enterprise picture. These capabilities are covered in separate technology-related perspectives, but it is important to understand that there are complementary and overlapping capabilities between SOA and the other technologies.

For example, SOA asset management and BPM asset management are quite similar and the same asset management solution can be applied to both. Business processes can be implemented by orchestrating Services and they can be exposed as Services for external consumption. BPM can leverage the Service discovery and routing capabilities of SOA to achieve that.

Similarly, SOA can take advantage of the messaging and eventing capabilities of the EDA infrastructure, and EDA can invoke the Services to perform an unit of work on the occurrence of an event.

3.1.7 SOA Capabilities and SOA Maturity

As the adoption of SOA increases, infrastructure will play a key role in advancing the level of maturity. It supports the activities and provides capabilities that are necessary to "raise the bar" in terms of what can be accomplished through SOA. Though these capabilities can be custom developed, it is generally more cost effective to buy the infrastructure rather than build it. Standards make it possible to mix and match best of the breed products across vendors and avoid vendor lock-in. They also make it possible, and even highly likely, for infrastructure to be acquired and deployed as the need arises as opposed to all at once.

Figure 3–2 SOA Capabilities and SOA Maturity

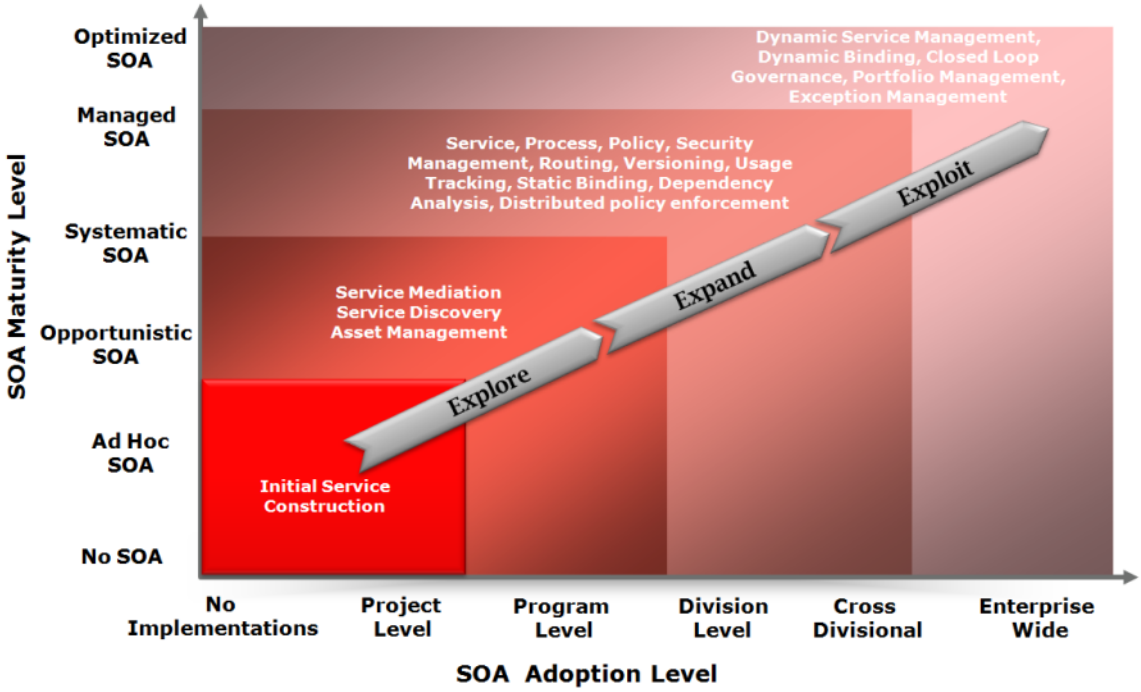


Figure 3–2 shows the typical set of capabilities required as the SOA maturity levels increase and SOA adoption becomes widespread in the organization. The required capabilities would also depend on the specific needs of the projects and Services being implemented. As shown above, maturity depends on the ability to define, manage, and optimize the SOA environment. It is also beneficial to ensure a standardized approach is taken as the breadth of SOA increases.

Initially, at the project-level, the benefits of SOA infrastructure are minimal. As SOA becomes adopted at the program level, the need for asset management, discovery, and mediation become important. The definition of Service engineering processes, categorization of assets into a taxonomy, and establishment of mediation patterns must emerge and become instituted; otherwise the proliferation of ad-hoc approaches will create chaos.

As SOA moves out across divisional boundaries, policies for security and management gain in importance. Centralized monitoring and management become more and more necessary in order to support expansion in a reliable and secure manner. The growth of the Service portfolio will promote building new business capabilities by assembling new Services. Service routing, versioning, usage tracking, dependency tracking and distributed policy enforcement capabilities would be most used at this level of maturity.

Further benefits can be realized as the environment is optimized for efficiency and performance. As organizations reach "SOA nirvana", they would be using the capabilities that assist with continuous improvement of the Services and business value. Dynamic Service binding, dynamic Service management, and closed loop governance are some of the capabilities that will be in high demand at that level of maturity. Infrastructure provides the means to gather key indicators to make optimization possible, either through manual or automated means.

3.2 Architectural Principles of SOA

Architectural principles establish the fundamental basis for architectural and design decisions for an implementation. All architectural decisions should be traceable back to some subset of these principles. Many architectural principles are specific to an enterprise, for example

- Compliance with the following set of enterprise and industry standards is required…
- All strategic Services will have a Web Services interfaces

A general list of principles of service-orientation (closely related to the benefits outlined previously) should be formulated from the following list of key attributes of a Service:

- Formal contracts
- Loose coupling
- Abstraction
- Reusability
- Autonomy
- Statelessness
- Discoverability
- Composability
- Granularity (modularity)
- Durability
- Interoperability
- Standards compliance (enterprise, business domain, IT industry)

Associated engineering and operational guides might include principles relating to

- Services identification
- Categorization
- Provisioning
- Monitoring and tracking

These lists can be used to develop a generic set of guiding principles as a basis for the general ground rules for development, maintenance, and execution of an SOA reference architecture.

The following is a set of generic architectural principles for Service definition and design:

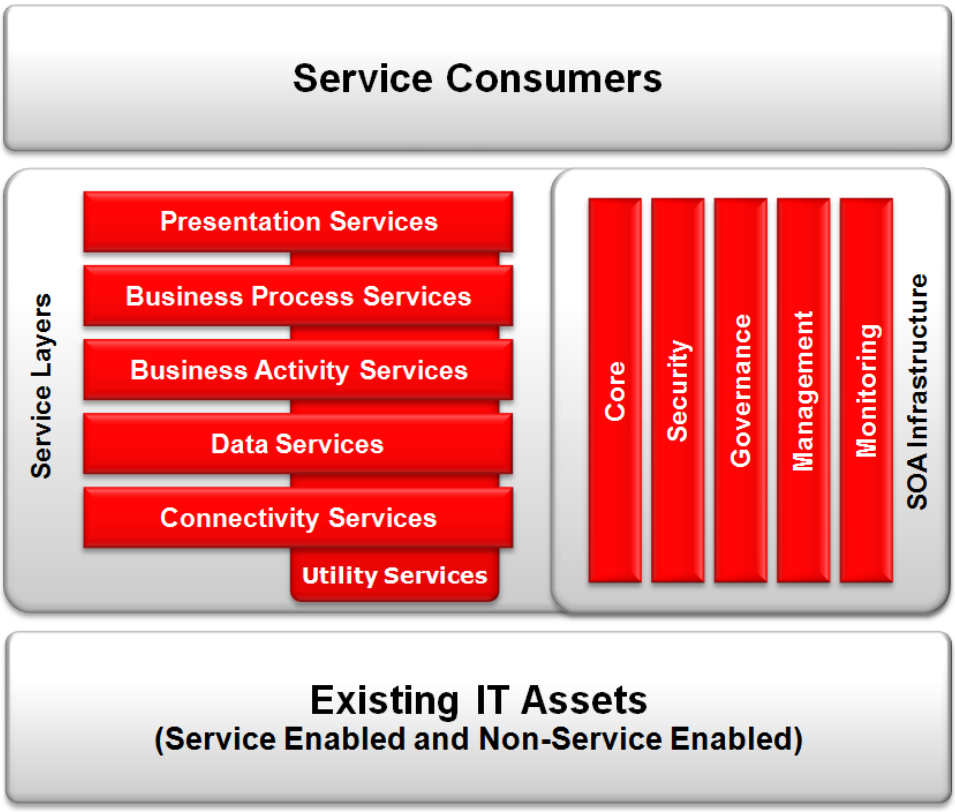
- **Service encapsulation** - A Service must perform a complete unit of work.
- **Service loose coupling** - Services maintain a relationship that minimizes dependencies between them.
- **Service contract** - Services adhere to a contractual agreement, as defined collectively by one or more Service description documents.
- **Service abstraction** - Beyond what is described in the Service contract, Services hide logic from the outside world.

- **Service reusability** - Logic is divided into Services with the intention of promoting reuse.
- **Service composability** - Collections of Services can be coordinated and assembled to form composite Services.
- **Service autonomy** - Services have control over the logic they encapsulate.
- **Service optimization** - All else equal, high-quality Services are generally considered preferable to low-quality ones ("quality" in this context is referring to business value, reuse, and other chosen measures of value).
- **Service discoverability** - Services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms.

3.3 Conceptual Architecture

As illustrated in Figure 3–3, the SOA consists of a set of layers and capabilities. The following subsections describe each layer of the architecture in terms of the capabilities it provides.

Figure 3–3 Conceptual Architecture



3.3.1 Service Consumers

Service consumers are, of course, consumers of Services, but more importantly they do not offer any SOA capabilities nor are they required to conform to SOA principles (assuming they are not also Service providers). Service consumers may be end-users, composite applications, or other systems in purely system-to-system interactions.

The special case of end user devices, such as laptops and computer terminals, represent the "SOA-at-the-glass" delivery channel employed by mashups, AJAX, fat Java clients, applets, etc.

3.3.2 Service Layers

The Service Layers section of the conceptual architecture represent categories of Services that serve different purposes. The Service categories are shown here as layers in the conceptual architectural model. Using this categorization, or layering of the architecture, we achieve greater "separation of concerns", such as separation of the presentation from the business logic, or the business processes from the purely technical infrastructure Services. The architectural layers also often employ different standards and technologies, however, Service categories are based on the type of capability provided and not the technology or product supporting it.

The Service Layers is comprised of various types of reusable Services following the Service-Oriented Architecture strategy. Services represent a modular approach to application architecture, where capabilities (operations and data) are exposed for common reuse via well-known interfaces.

Consumers of Services are often unaware of the physical location of a Service (endpoint). Instead, they use SOA infrastructure components to discover Service endpoints and mediate requests.

Services may be developed from a number of sources, such as exposing operations or data from custom or legacy applications, packaged applications, databases, etc. They may also be developed from scratch using popular application infrastructure platforms such as J2EE,.NET, and Tuxedo.

Refer to the *ORA SOA Infrastructure* document for more details regarding the Service categories.

3.3.3 SOA Infrastructure

Much the same way that container-based infrastructure provides core services to J2EE based applications, SOA infrastructure provides the enabling capabilities required to realize and mature a SOA deployment. SOA infrastructure provides the technology that is a common need among all SOAs and allows enterprises to focus on enabling business capabilities within their SOA, rather than building enabling technology.

A key capability that SOA infrastructure enables is Service enablement. Service enablement is a key differentiator of Service implementation (as opposed to application implementation). The difference between enablement and the rest of the implementation is that Service enablement is accomplished through infrastructure, versus deployment-based artifacts originating from code. The purpose of Service enablement is to separate core QoS concerns for a Service.

In addition, the SOA infrastructure provides value by acting as an intermediary and mediator between consumers and providers. For example, intermediaries can bridge the technology gaps between two parties. In addition the SOA infrastructure enables loose coupling by decoupling the Service consumer's dependence on the Service provider. With loose coupling a Service provider can be versioned, physically moved, or replaced without affecting the Service consumer. The SOA infrastructure routing capability enables the loose coupling required for SOA by providing the ability to control where a message is sent or which Service endpoint is invoked.

3.3.4 Existing IT Assets

Existing IT assets can be grouped into two distinct groups - Service enabled and non-Service enabled. Service enabled assets, sometimes referred to as "SOA-enabled or service-enabled applications", may also be a Service consumer.

Service enabled applications are essentially applications deployed with embedded Services; they can be subcategorized into various types such as packaged applications, partner applications (which includes SaaS, cloud, etc.), and custom applications.

In contrast non-service enabled assets (also referred to as legacy or traditional applications) do not provide Services. Instead the SOA infrastructure can be utilized to expose this functionality.

4

Technologies and Standards

A key benefit to SOA, and most likely reason to succeed, is that many companies have worked together to define standards for interoperability. Though there are many standards, versions, and options, the result of this work is still beneficial. While previous attempts to promote interoperability fell short due to proprietary extensions and implementations, SOA has a much better foundation to work from in this regard than perhaps any previous wave of technology. For this reason, standards support should be considered a priority when defining the SOA reference architecture and the infrastructure therein.

Several industry standards have been defined around SOA and SOA infrastructure technologies. Given the unique nature of the SOA infrastructure and the ability to create a best of the breed SOA infrastructure using disparate technologies, it is imperative to take a standards based approach in building the infrastructure. When an organization chooses the tools and technologies for its SOA implementation, it needs to consider the extent of standards support in these technologies and ensure that it will meet the requirements of that organization.

Unfortunately there are many standards and simply selecting them all is not a path to automatic success. Many standards are not yet complete, or more importantly, not yet adopted by the tools and infrastructure vendors: in this case an unsupported standard will do nothing for short term interoperability (and in fact, if it is ultimately not adopted, then it also offers no benefit in the long term).

So it is important to select the right standards for any given enterprise. This leads to the question, "Why would standards selection vary between enterprises?" There are a number of relevant considerations when selecting standards, including the industry domain and existing methods and competencies that may already in place within an organization.

Standards may originate from various sources:

- IT industry standards (e.g. W3C XML, OASIS WS-*);
- Business industry standards (e.g. HL7, …);
- Enterprise standards (engineering standards).

It is important to distinguish standards from policies or regulations (such as HIPA and Sarbanes Oxley): in the case of "enterprise standards", it is important to distinguish from principles, practices, and guidelines (such as "Services must not cross domain boundaries below enterprise level") which may be less strictly defined.

Once we have determined what standards may be available to a domain another consideration is which standards are relevant to the architectural style of the organization's engineering practices. The next section outlines the key standards for

SOA while the section that follows it deals with the different architectural styles that have a bearing on standards selection.

4.1 SOA Standards

This section introduces standards relevant to SOA, including Service design, implementation, deployment, and management; it does not cover general software engineering and IT standards, nor does it include industry specific standards.

The primary benefits of adopting standards are interoperability, technology independence, and rapid integration. By avoiding proprietary protocols, formats, etc. and adopting standards-based strategies, the enterprise can avoid vendor lock-in, simplify exchange of information between applications and partners, and take advantage of readily available development skills.

Standards are available in many forms and for a variety of different purposes. Core SOA related standards recommended and described in this document are:

- Java API for XML Web Services (JAX-WS)
- Business Process Execution Language (BPEL)
- A collection of Web Services standards (WS-*)
- Simple Object Access Protocol (SOAP) with attachments and Message Transmission Optimization Mechanism (MTOM) (implemented as SOAP with Attachments API for Java (SAAJ)
- HyperText Transfer Protocol (HTTP)
- Universal Description, Discovery, and Integration (UDDI)
- Service Component Architecture (SCA)
- Apache's Web Service Invocation Framework (WSIF)
- Web Service Interoperability (WS-I)

Some examples of industry specific standards that may be relevant to SOA, but excluded from this document for brevity, include

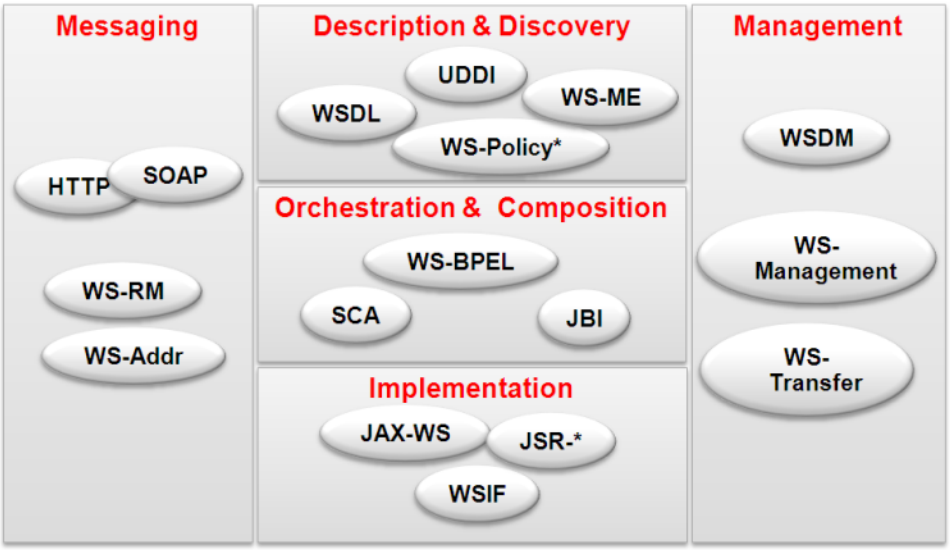
- HL7 (XML messaging standard for Healthcare for the exchange, integration, sharing and retrieval of electronic health information)
- HRXML (a library of XML schemas developed by the HR-XML Consortium, Inc. to support a variety of business processes related to human resource management)
- Telecoms IMS (IP Multimedia Subsystem, 3GPP Technical Specification TS 22.228) architectural strategy and standards for decoupling of the technical and organizational constraints of former telecommunications strategies.
- OASIS ebXML work, such as, CPPA (Collaboration Protocol Profile and Agreement) which describes how trading partners engage in electronic business collaborations through the exchange of electronic messages

Standards for SOA can be categorized into four areas of concern:

1. Orchestration and composition;
2. Description and discovery;
3. Messaging;
4. Implementation.

These categories of standards are shown in the diagram along with some of their associated standards abbreviations.

Figure 4–1 Categorization of Standards with respect to SOA



The categories and their core standards shown in Figure 4–1 are described in following sections.

4.1.1 Orchestration and Composition

The Enterprise Service Bus (ESB) is commonly thought of as the primary platform for Service orchestration, however, the ESB has no standard definition and although it typically supports many standards internally the ESB platforms are proprietary. The primary standards for Service orchestration and composition are listed below.

- WS-BPEL (formerly known as BPEL4WS and commonly abbreviated to BPEL)
  - BPEL is a common standard for assembly of processes (business and system) based on Services and defined using WSDL. A BPEL orchestration might contain a single activity, simply invoking a Service, or many activities to accomplish long running tasks involving complex transformations, joins, loops, with synchronous and/or asynchronous processing.
  - BPEL infrastructure should accommodate (in addition to current BPEL specifications implementation) manual task workflow and may support alternative Service exposure (besides WSDL with its implicit SOAP/HTTP binding), such as, Apache's Web Service Invocation Framework (WSIF)
- Java Business Integration (JBI) (JSR-208)
  - Normalized message routing for standardized Service assembly ensuring loose coupling between bindings and Services.
  - May be used to deploy Service assemblies on top of BPEL and ESB infrastructure.
- Service Component Architecture (SCA)
  - A programming model for Service based application development.

- Defines Service component and assembly models (unlike WSDL).
- Enables specification of dependencies on other Services and policies, such as, transaction, security, reliable messaging, etc.
- Supports a number of data connectivity standards including, JDBC, Java Persistence API (JPA), and Service Data Objects (SDO).

BPEL is the primary focus here for orchestration and composition. It is an OASIS standard for a Web Services orchestration language: it provides property-based message correlation, XML and WSDL typed variables, XPath for expressions and queries, structured programming constructs if-then-else, while, sequence (ordered execution) and flow (parallel execution), fault- compensation- and event-handlers. BPEL is best suited to long-running (business) processes, providing correlation between running processes, orchestration of Web Services, and composition of Services.

Although JBI and SCA are listed here they appear in the implementation category in the diagram above: this may seem a little confusing at first, but it is presented this way because they provide composition-like capabilities at a software level. This software level of composition is commonly referred to as Service assembly which is described in the next section.

4.1.1.1 Service Assembly

There is no strict definition of the term **Service Assembly**, but is generally used to refer to low-level Service composition.

There is a proliferation of standards work describing the components and patterns of Service assembly. These standards may be broken down into a number of categories:

- Business process definition, represented by BPMN, BPML, and others.
- Service orchestration, represented by BPEL, XLANG, WSFL, and others.
- Workflow, represented by XPDL, BPXL, and others.
- Service choreography, represented by WSCI and others.
- Service container technologies, represented by JBI, SCA, and others, that have interactions with orchestration and choreography.
- Portlets, represented by WSRP and JSR 168, the Java Portlet API Specification.

4.1.2 Description and Discovery

Service description and discovery refers to the standards based techniques for advertising and finding a Service according to its capabilities.

WSDL is the primary means by which a Service is described. A WSDL representation of a Service has separate abstract and concrete parts. The WSDL abstract part describes the Service interface including its operations and message structure; its concrete part describes protocol bindings (the language e.g. SOAP) and endpoints (where to find it).

UDDI describes a mechanism for registering a Service to enable it to be discovered by other consumers. Service discovery can occur at either design-time or runtime. During the construction (design-time) of a composite application, for example, a Service registry might be searched for Services and operations matching specific requirements (this typically corresponds to the abstract portion of the WSDL document). At runtime the UDDI registry is generally used to find the location (endpoint) of the current version of a Service.

WS-Policy and WS-MetadataExchange are used to further refine the description of a Service. These are described in more detail under the Web Service section of this chapter.

4.1.3 Messaging

Messaging describes the protocols and formats of the information exchanged by Services.

HTTP is an application layer protocol at the top of the Internet Protocol (IP) stack. HTTP was used initially for internet browser interaction, but has become more wide used to support a variety of internet functions; it is the most common protocol binding used for Web Services SOAP messaging.

SOAP is a protocol specification for exchanging structured information in the implementation of Web Services.

WS-ReliableMessaging and WS-Addressing are enhancements to message exchange descriptions. These are described in more detail under Section 4.2.4, "Web Services Standards" of this chapter.

4.1.4 Implementation

Enterprise software development standards for SOA naturally focus on specific implementation platforms. Java, for example, provides many implementations of standards as extensions to the language or the Java Enterprise platform. These Java implementation standards are typically described by Java Specification Requests (JSR) maintained by Java Community Process (JCP).

The collection of Java standards implementations, focusing on the Web Services architectural style (see next section for more information) is referred to as Java Web Services (JWS). JWS is not a standard itself, but refers to implementations included with Java Enterprise Edition 5 and Standard Edition 6. The following is a outline of the JWS standards and their implementations:

- JAX-WS 2.0 (successor to JAX-RPC / XML schema data binding removed to JAXB) WSDL-to-Java mapping (with support for REST endpoints) (JSR224)
- JAXB 2.0 XML schema to Java data binding (JSR222) (separated from former JAX-RPC)
- WS-Metadata (annotation for definition and deployment) (JSR181)
- SOAP with Attachments API for Java (SAAJ) (JSR67)
- WS for Java EE 1.2 (WSEE) (JSR109)
- JAX-R interface for UDDI repositories (included with J2EE 1.4 and Java EE 5)

4.2 SOA Implementation Architectural Styles

The concepts of encapsulation, separation of concerns, layered architectures, and distributed computing have evolved over the last two decades; these are the core elements of SOA implementation. Due to the history of SOA's evolution some people claim to have been doing SOA long before the term was first coined. In reality the true benefits of SOA can only be realized through adoption of common standards and suitable infrastructure. For example, CORBA promised many of the benefits of today's SOA, but it fell short in fundamental aspects, such as platform independence, not to mention more substantial SOA opportunities like Service composition.

4.2.1 REST

The unprecedented success of the Internet is largely attributable to Internet Protocol (IP) and within that Hypertext Transfer Protocol (HTTP), the most common application layer protocol in the four-layer TCP/IP model.

HTTP provides a stateless, platform-independent protocol for the exchange of information over a network; it is a standard for a request/response dialogue between clients and servers.

From the definition of HTTP (or more accurately its principle author, Roy Fielding) an architectural style emerged called Representational State Transfer (REST). Core to the concept of REST is a resource which it defines as any item of information or capability addressable on a network by a global identifier (its URI in HTTP terms). The REST architectural style describes any simple interface which transmits domain-specific data over HTTP without involving higher-level messaging, such as, SOAP (see Web Services Architecture below).

REST shares architectural principles with HTTP:

- Application state and functionality are abstracted into resources.
- Every resource is uniquely addressable using a universal syntax for use in hypermedia links.
- All resources share a uniform interface for the transfer of state between client and resource, consisting of:
  - A constrained set of well-defined operations.
  - A constrained set of content types, optionally supporting code on demand.

While Fielding's REST was never intended to provide "Web Services" in the context of SOA, some believe that its simplistic model will lead to greater success than the substantially more complex approach of Web Services Architecture (see Section 4.2.3 below).

While REST shares some of the features that made the Internet so enormously scalable, its simplicity evades some of the more important aspects of today's business computing. It does not provide standardized mechanisms for:

- describing the quality of its Service (QoS, including reliability, availability),
- transaction management,
- security.

While REST's simplicity supports rudimentary mash-ups by almost any user, it does not easily and comprehensively support the rapid composition of complex business processes.

There are some circumstances, however, where the simplicity of REST makes it an appropriate choice. For example, in the public domain where consumers are unlikely to be sophisticated enough to conform to richer standards like Web Services. Similarly, environments that are not transactional or do not require high levels of security may be suited to a REST architectural style.

The choices of architectural style are also not mutually exclusive: REST, Web Services, and potentially other styles can coexist, albeit at a higher cost in infrastructure and engineering skills. Support for REST is also starting to appear in Web Services standards: WSDL 2.0, although not yet widely implemented (by tools vendors), has broadened its binding specification to include all HTTP methods, thus implicitly including support for RESTful Web Services.

4.2.2 Remote Procedural Call (RPC)

Fundamentally RPC is a means of calling a method or function on another node in a distributed computing environment. Originally RPC's were not standardized in any way and typically comprised mysterious, idiosyncratic, poorly documented Unix socket code. As such the basic RPC offers none of the benefits of SOA. Early examples of standardized RPC's included Sun's RPC, Microsoft's DCOM, Java's RMI, OMG's CORBA, ODBC/JDBC, and SQL*Net. More recent RPC's include XML-RPC, and HTTP.

Although HTTP is itself a form of RPC (its limited set of commonly used method calls are GET, POST, PUT, DELETE), REST is characterized by the exchange of resources (nouns) which differentiates it from RPC's exchange of commands (verbs).

RPC is now considered an application layer protocol in the TCP/IP model.

4.2.2.1 XML-RPC and SOAP

The cornerstone of today's platform and technology independence is eXtensible Markup Language (XML). It is a language definition that yields conversations so verbose that only modern networks can accommodate it. Due to its simplicity, platform independence, and extensibility a series of rapid steps were taken to exploit XML. These are outlined in the following paragraphs.

XML-RPC is an unusually simple RPC standard using XML for its payload.

Simple Object Access Protocol (SOAP) evolved from XML-RPC concepts. SOAP is a protocol specification for exchange of structured information using XML and TCP/IP application layer protocols (typically RPC and HTTP) to form the basis for Web Services message exchanges and is therefore a common enabling technology for SOA.

Asynchronous JavaScript and XML (AJAX) refers to a group of Rich Internet Application (RIA) development techniques that exploit the XMLHttpRequest method of the browser Document Object Model (DOM) using XML-RPC. AJAX is an enabling technology in the loosely defined Web 2.0 realm. Although it is not directly related to SOA, Web 2.0 does offer some composition techniques typically referred to as "mashups".

4.2.3 Web Services Architecture (WSA)

Many people equate SOA with Web Services or even BPEL. Web Services, BPEL, and a long list of associated specifications describe the commonly used technologies developed in support of SOA. However, it is important to separate the implementation technologies from the broader subject of the architecture itself.

The World Wide Web Consortium's (W3C) Web Service Architecture (WSA) provides a definition of Web Services. The WSA requirements document provides background and purpose to the WSA and defines a Web Service that we can use to contrast the definition of a SOA Service. The following definition is taken from the W3C WS Glossary:

"A Web Service is a software system identified by a URI [RFC 2396], whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web Service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols."

The WSA breaks down Web Services into four related models:

- Service-oriented model,

- message-oriented model,
- resource,oriented model,
- policy model.

The models describe the aspects of action, message, resource, and policy respectively.

Many other RFC's underpin the fundamental definition of Web Services but, more relevant to a SOA RA, is the catalog of Web Services standards extensions that support interoperability in the infrastructure. These include:

- WS-I,
- WS-Security,
- WS-Addressing,
- WS-SecurityPolicy,
- WS-ReliableMessaging (and WS-ReliableMessagingPolicy),
- WS-SecureConversation,
- WS-Security,
- WS-Transactions (plus WS-AtomicTransaction and WS-BusinessActivity),
- WS-Trust.

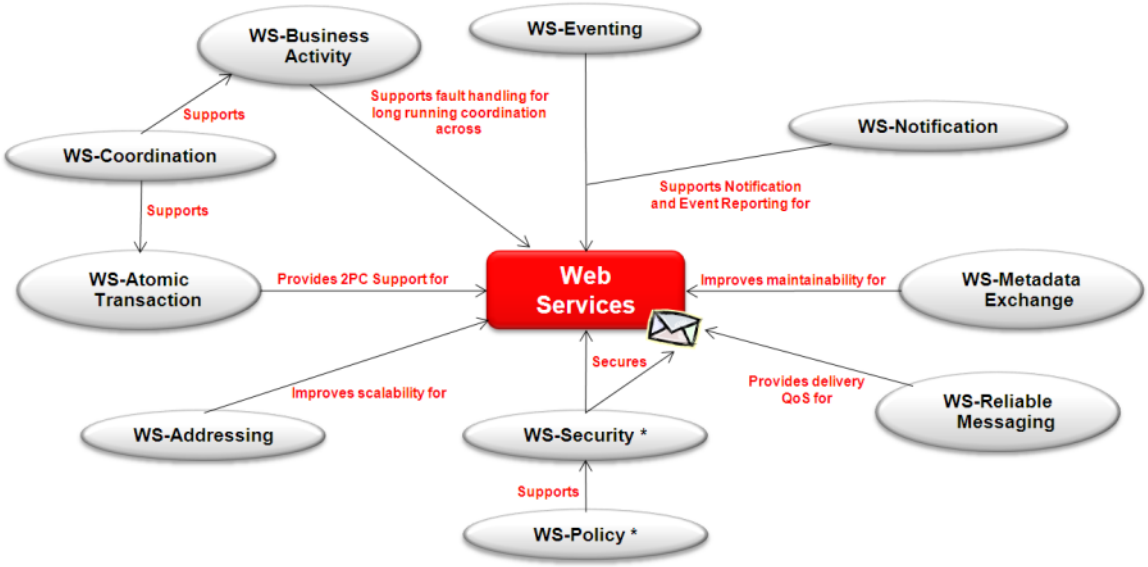
One of the major inhibitors of SOA adoption, at least in the beginning, was the inability to achieve certain levels of QoS through a Service-Oriented Architecture. The initial release of specifications utilizing Services in the form of Web Services had very weak support for various standard levels of QoS-based requirements.

4.2.4 Web Services Standards

The second generation of Web Services specifications (commonly referred to as WS-*) has a large focus on enabling QoS within an SOA environment, which is one of the reasons Web Services has become the primary way to expose Services. This section presents a high-level description of the Web Service second generation specifications. The standards described in this section do not constitute an exhaustive list, but instead focus on those specifications that are recommended by Oracle.

A high level view of the QoS supporting specifications is illustrated below.

Figure 4–2 Web Services Specifications



The following sections provide an outline of the standards specifications in Figure 4–2 above.

4.2.4.1 Web Services Transaction (WS-TX)

Web Services Transaction (WS-TX) is a collection of related specifications whose purpose is to define a set of protocols to coordinate the outcomes of distributed application actions. Transaction coordination is critical at various levels of Service interactions. Strategies, such as the classic two phase commit, can be used for short-lived Service orchestrations; but more complex methods are often needed in long running business processes in which resources must be committed before the final outcome is know: in these circumstances **compensating transactions** may be required. WS-TX provides a framework to handle these different scenarios.

WS-TX specifies an extensible framework for developing coordination protocols through the Web Services Coordination specification, which in turn, is a framework used by two coordination types: atomic transaction (WS-AT) and business activity (WS-BA)

4.2.4.1.1 WS-Coordination

WS-Coordination enables scenarios in which a number of applications are required to reach consistent agreement in order to complete an operation. This form of coordination is critical to business process orchestrations.

The WS-Coordination framework enables an applications to propagate an activity between Services and to register for coordination protocols. The framework enables existing transaction processing, workflow, and other systems to hide their proprietary protocols and to operate in a heterogeneous environment.

WS-Coordination defines the structure required to propagate context between cooperating Services, however, it does not provide the coordination for transactions between Service: it is only a coordination framework requiring other specifications, such as WS-Atomic Transaction and WS-BusinessActivity, for this purpose.

4.2.4.1.2 WS-AtomicTransactions

WS-Atomic Transactions provides agreement coordination protocols for the following transactional types:

- completion,
- volatile two-phase commit,
- durable two-phase commit.

WS-AT is used in applications that require consistent agreement on the outcome of short-lived, distributed operations.

4.2.4.1.3 WS-Business Activity

WS-Business Activity is used in applications that require consistent agreement on the outcome of long-running distributed activities. In particular WS-BA provides compensating transaction support to the BPEL specification.

4.2.4.2 WS-Addressing

This specification provides support for enabling messages as autonomous units of communication. This specification allows for the inclusion of the following:

- From Address,
- Return Address,
- Return Address if unable to delivery to requested return address.

WS-A provides transport-neutral mechanisms to address Web Services messages. It defines a set of abstract properties using an XML representation to facilitate end-to-end addressing of endpoints in messages to enable systems to support message transmission through networks endpoint managers, firewalls, and gateways.

This is another specification important to BPEL business process orchestrations and other messaging infrastructure products such as the Enterprise Service Bus.

4.2.4.3 WS-Policy *

This set of specifications is an enabler for other specifications and is supported by most Web Service technology providers. WS-Policy * is used by other WS-* specifications to pass metadata and enable dynamic policy decisions. Its impact on QoS is that it restricts the delivery of messages to those consumers that conform to the predefined policy rules. This specification has been or is being adopted mostly by vendors that provide for WS-Security based specifications (in particular WS-SecurityPolicy).

4.2.4.4 WS-Security *

The WS-Security family of services provides a security framework for securing Web Services and messages passed in an SOA. This specification is being adopted by most Service infrastructure vendors.

4.2.4.5 WS-Reliable Messaging

This specification is designed to support the delivery-based QoS for messages (e.g. guaranteed delivery, at most once). Partial support for this specification is provided by most Service infrastructure vendors.

4.2.4.6 WS-Meta Data Exchange

This specification enables the ability for Services to communicate the information needed in order to communicate and interact with one another. Requestors can then verify that the correct metadata is in fact being used for their planned message exchanges. This can increase QoS because it tends to avoid a multitude of maintenance problems associated with Service contract changes. This specification has not seen any adoption to date.

4.2.4.7 WS-Eventing and WS-Notification

Integration is a key benefit of SOA largely as a result of its platform independence and open standards. EAI, MOM, and other earlier generation architectural strategies have been generally adapted to offer their capabilities through standards compliant, discoverable interfaces. Many of these adopt the Web Service model (benefiting from technology independence and rapid development) but few ultimately conform to a true SOA definition.

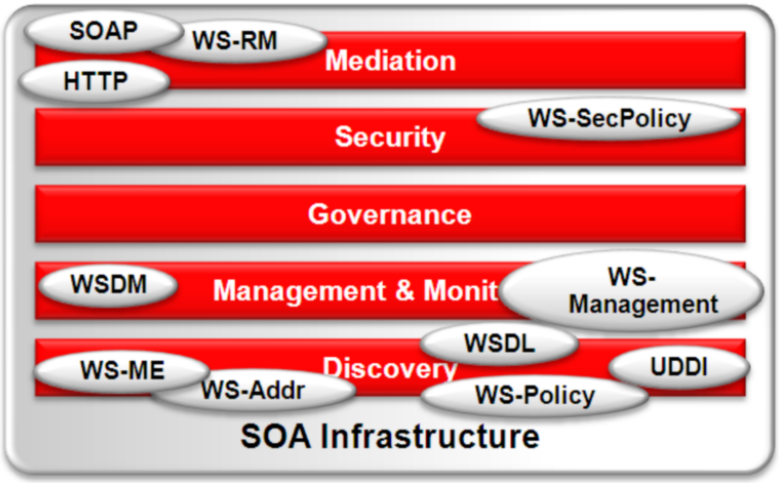
4.2.4.8 WS-I

WS-I is an industry consortium (referred to as the Web Services Interoperability Organization) whose charter is to promote interoperability across the Web Services specifications. The WS-I organization does not define standards for Web Services; instead it creates guidelines and tests for their interoperability.

4.2.5 Infrastructure Standards

Looking back at the earlier standards discussion we can now map the relevant standards to the infrastructure. The following diagram maps the relevant standards to the appropriate SOA infrastructure capabilities highlighted in the SOA conceptual Architecture.

Figure 4–3 SOA Standards Mapped to SOA Infrastructure Capabilities



It is interesting to note the core SOA standards focus on certain aspects of the infrastructure. There are, however, many more standards underpinning those shown in Figure 4–3, such as XML Digital Signatures associated with WS-SecurityPolicy; many others are not shown because they are not exclusive to the SOA domain.

5
Summary

SOA is often equated with Web Services, so one of the primary objectives of this document is to emphasize how the two are different, although also commonly complimentary. Web Services is technology based on a set of implementation standards and having its own architecture (**WSA**), whereas SOA is an Enterprise Architecture (EA) strategy. In this broader scope of SOA there is a technology agnostic definition of a Service, under which various technology strategies may be employed.

Services within the SOA definition arise in many forms: encapsulated legacy application functions (non-Service enabled assets) exposed via Connectivity Services, legacy functionality embedded in Services at any architectural level, legacy enablement (Service enabled assets), composition of Services, etc. Services may also be the result of new software development where new requirements are identified, but this is not the place to start when a common goal of SOA is to maximize the value of existing assets.

The focus of this document is architecture, and therefore, the approach to defining SOA has taken the path of dissecting a Service into its constituent parts, the contract, interface, and implantation. These parts were further dissected and defined to ultimately produce models, categorization schemes, and architectural principles. It is important to remember, however, that an IT strategy requires more than a reference architecture to execute it in an IT environment. Methods and procedures are needed to fulfill the broader aspects of an enterprise strategy, such as business justification, governance, enterprise engineering approach, etc., but these topics are beyond the scope of this document. Due to the typical complexity of enterprise-wide engineering projects, with broad goals like maximizing IT flexibility and responsiveness to changing business demands, a reference architecture plays a critical role in harmonizing the use of standards, establishing a common vocabulary, and generally applying the guidelines for seamless interaction between otherwise siloed business units.

Finally, and most importantly, this document has defined the SOA foundation for the Oracle Reference Architecture and has established the ground rules for both the subsequent ORA documents and associated Enterprise Technology Strategies.

A

Other Service Models

Because of the widespread attention that SOA has received, there are a variety of standards emerging around SOA. Two of particular interest are the ***Reference Model for Service Oriented Architecture*** (SOA RM) from the OASIS standards body and the ***Service oriented architecture Modeling Language*** (SoaML) from the Object Management Group (OMG).

This appendix provides a very brief description of SOA RM and SoaML and compares and contrasts them to the Service definition discussed in Section 2.2 and formalized by the Service meta model presented in Section 2.4.

This appendix uses capitalization to distinguish ORA defined terms from other industry terms. When used as a noun, the word service is capitalized when it means the ORA SOA Service. When used as an adjective, the word service is not capitalized unless it is part of an ORA defined term. For example, service consumer is not capitalized since ORA does not specifically define this term; whereas Service Contract is capitalized because ORA defines this term as an essential aspect of a Service.

This usage of capitalization is particularly important when comparing ORA to other definitions since the other definitions use the same words (e.g. service contract) but they do not mean the same thing. The ORA terms are capitalized; whereas the other terms (using the same words) are not.

A.1 SOA RM

The SOA RM is a very high level definition of terms intended to provide commonality when discussing or describing SOA. The reference model is intentionally devoid of specific technologies, implementations, or concrete details. The SOA RM specification is available from OASIS at: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

A.1.1 Definition of a Service

The SOA RM describes the constituent parts of a service as:

- the service description,
- the execution context of the service,
- the contracts and policies that relate to services and service participants.

A.1.1.1 Service Description

The service description represents the information needed to use the service. This includes the service interface as well as descriptions of what the service operations perform. The service description also includes the information model used as input and output by the service.

A.1.1.2 Execution Context

The execution context is the totality of the interaction between the service consumer and the service provider including all infrastructure elements and process entities. Essentially, the execution context is the path between the service consumer and service provider.

A.1.1.3 Contracts and Policies

Contracts and policies define the conditions for use of a service. A policy is a condition defined by one participant; whereas a contract is an agreement between two participants.

A.1.2 Discussion

The SOA RM definition of a Service is similar to the one provided by ORA. There is no glaring discrepancy between the two concerning the definition of a Service. However, the two differ on the specifics of the constituent parts of a Service. The SOA RM describes three constituent parts of a Service just like ORA does. ORA breaks out the Service Interface as a first class facet of a Service; whereas the SOA RM lumps the service interface in with the service description.

ORA separates the human-readable, textual aspects of the Service from the technical aspects. The textual aspects are the Usage Agreement and the Service Contract. The technical aspects are the Service Interface and the Service Implementation. The SOA RM does not separate them. Both the service description and the contracts and policies include both technical and textual contents.

The execution context of SOA RM relates closely to the Service Implementation aspect from ORA. The execution context is specifically focused on the path between the service consumer and the service provider; whereas the Service Implementation focuses on the computing elements that provide the capabilities offered by the SOA Service.

The SOA RM does not include the concept of the Usage Agreement. It appears that in SOA RM terms the Usage Agreement would be classified as a type of contract.

A.2 SoaML

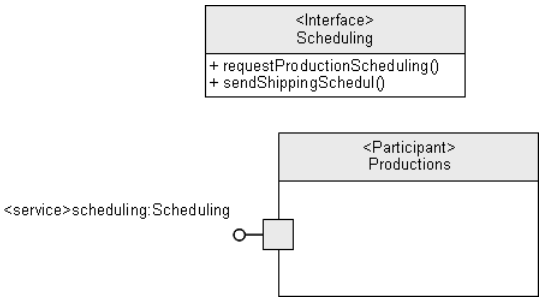
SoaML is designed to support service modeling and design providing sufficient detail to support a model-driven development approach. SoaML is based on UML 2.0 specification and provides a fully defined meta-model that uses the extension mechanisms provided by UML 2.0. The SoaML specification is available from the Object Management Group at: http://www.omg.org/spec/SoaML/Current

A.2.1 Service Modeling

There are several different UML stereotypes defined by SoaML that can be used to model services. This section only describes a few of these stereotypes that are of particular interest when comparing with ORA terms.

Service modeling in SoaML is based on interfaces and participants. Figure A–1 shows a simple example of a service.

Figure A–1 Service Modeling in SoaML



This figure shows one service "Scheduling" provided by the participant "Productions". This example uses the base interface stereotype that is part of UML 2.0. SoaML also defines a service interface that is composed of two interfaces and a behavior. SoaML also includes a service contract stereotype (an extension to the UML collaboration) that illustrates how two or more services interact.

A.2.2 Discussion

The interface stereotype from UML 2.0 maps directly to the Service Interface as defined by ORA. Similarly, the participant stereotype maps directly to the Service Implementation defined by ORA. SoaML does not address either the Service Contract or the Usage Agreement defined by ORA.

SoaML defines a service interface stereotype. Unfortunately, this definition is a radical departure from what interface has traditionally meant in software engineering. The service interface stereotype appears to have been created in a misguided attempt to handle conversations between participants via interfaces. Conversations are not new to software design. Conversations can be modeled using base UML 2.0 for both classes and components, so why an entirely new construct is required for service modeling is a mystery.

SoaML defines a service contract stereotype. However, this stereotype is mis-named and does not meet the needs of a Service Contract as defined by ORA. The service contract stereotype should be called a service collaboration since what is modelled is the interaction between two or more services. The service contract stereotype is an optional model for services in SoaML whereas the Service Contract in ORA is an essential aspect.