Oracle® Reference Architecture

SOA Foundation

Release 3.1

E14484-03

September 2010

ORACLE®

ORA SOA Foundation, Release 3.1

E14484-03

Primary Author: Mark Wilkins

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

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

Contents

3 Technologies and Standards

4 Packaging and Deployment

5 Summary

List of Figures

Send Us Your Comments

ORA SOA Foundation, Release 3.1

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.

A logical architectural framework is provided as the foundation for ORA describing a layered Services model and supporting infrastructure. While some product mapping and infrastructure are outlined here a more detailed coverage is provided by other documents in the ORA series.

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 architecural 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 designers 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, its benefits, and the key characteristics of Services.

Chapter 2 - defines a logical view of SOA and introduces formal definition of a Service along with a UML meta-model for a Service.

Chapter 3 - explores standards and technologies.

Chapter 4 - looks at what's new in the deployment of SOA, compared to other project deployment strategies.

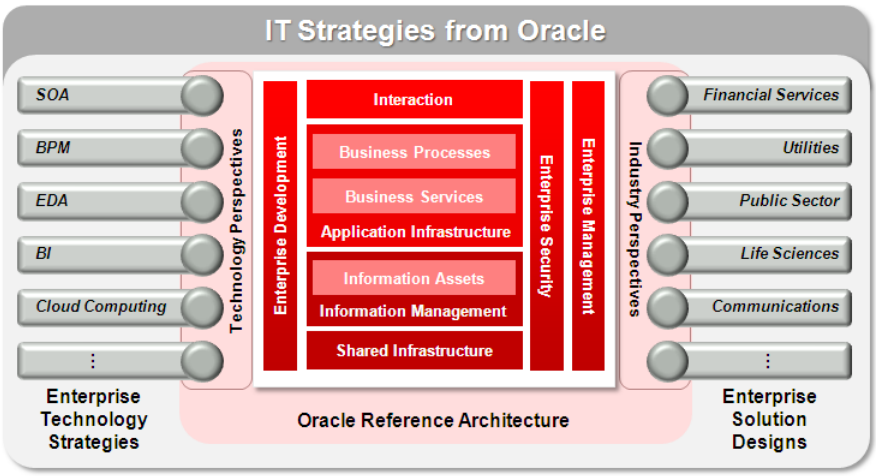
How to Use This Document

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 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

What is Service Oriented Architecture?

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 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.

1.2 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.

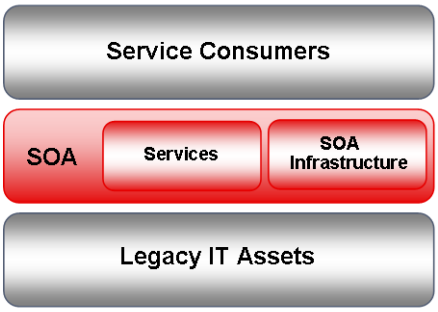
While SOA tackles all these major concerns it is important to note that it is not a panacea, nor is it a replacement for the many foregoing technologies and strategies. In some respects the underlying concepts have been around since the early days of distributed computing, for example in the form of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

1.3 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 Service 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 over time

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

1.4 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 & 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 above) 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

1.5 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.

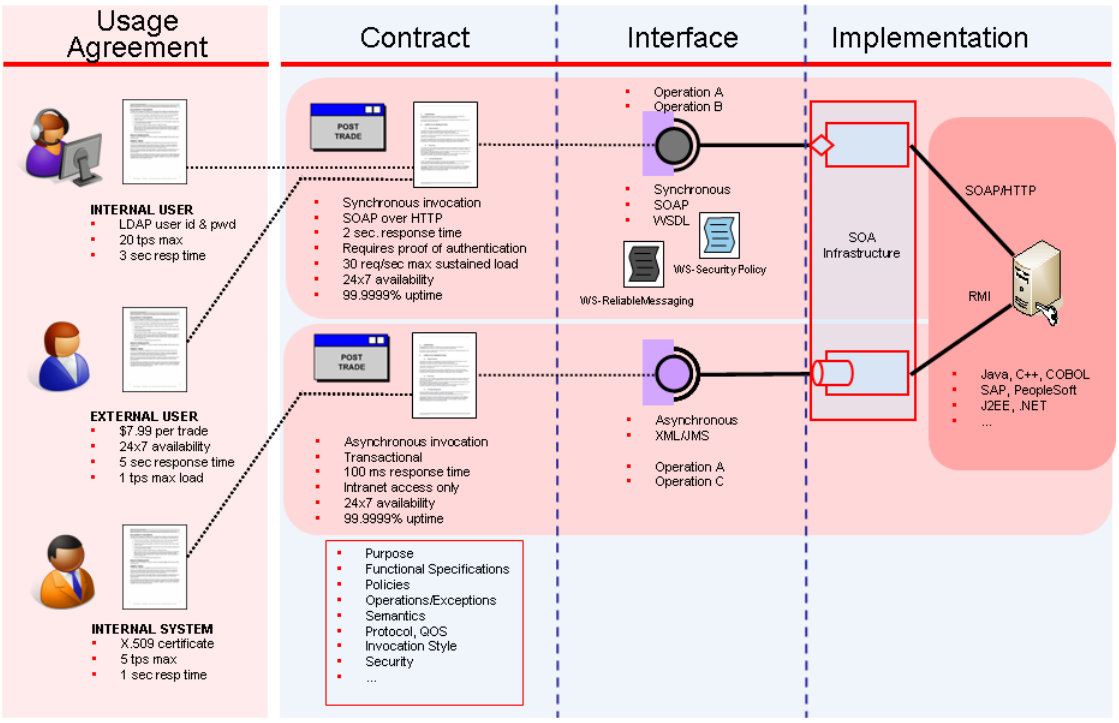
In addition to request and response data, Service interfaces may need to support additional information, such as security data, endpoint information, and binary attachments. They may also need to support certain quality of service constructs such as reliable messaging and secure conversations. For this 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 (see Section 3.2, "SOA Implementation Architectural Styles" for alternative implementation technologies).

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 1–2

Figure 1–2 Service Illustration



In addition to the service contact, interface, and implementation discussed so far, Figure 1–2 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

Architectural Models

Models are an important means of conveying ideas and representing architectural concepts. Models are useful for many different stakeholders and this reason they take many difference forms. A model typically used in executive level discussions to represent ideas about the business and its activities without the complexity of how it might be achieved is referred to as a Conceptual Architecture model. A similar, but generally more detailed model is the Logical Architecture. The Logical Architecture typically used as a high-level view by architects to represent a complete system and its architectural approach, but without implementation details.

The following sections describes various architectural models, starting with a logical representation, building on the concepts introduced in the Conceptual Model.

2.1 SOA Logical Architecture

A Logical Architectural view is used in this section to represent categories of Services that serve different purposes. The Service categories are shown here as layers in the Logical 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 Logical 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.

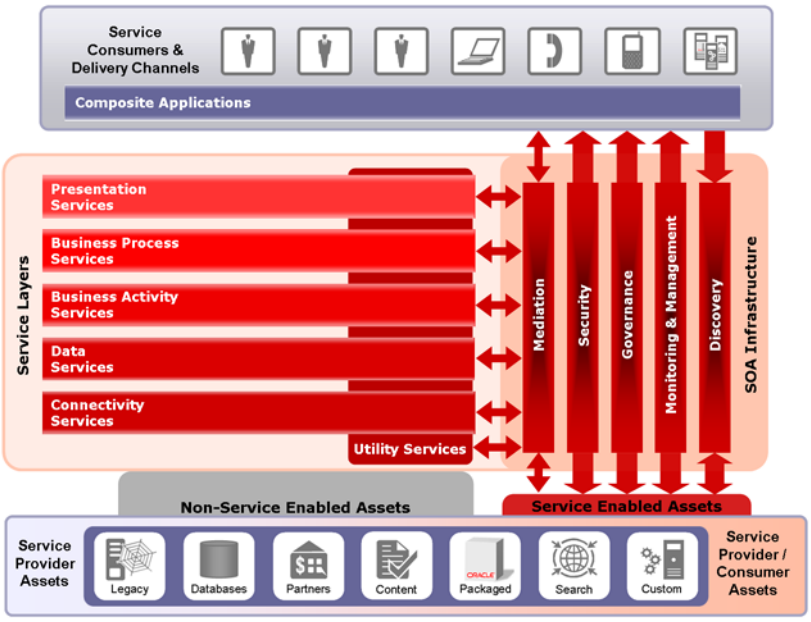
The Services Layer is comprised of various types of reusable Services following the Service-Oriented Architecture strategy. Services represent a modular approach to application architecture, where capabilities (operations and data) are exposed for common reuse via well-known interfaces.

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. Therefore the Services Layer will often be dependent upon an Application Infrastructure Layer and a Data Management Layer.

Since Services can be leveraged by a multitude of consumers, they must possess qualities of high availability and scalability. These qualities may either come from the Application Infrastructure itself, or by means of Virtualization (see Virtualization Layer).

Figure 2–1 SOA Logical Model



2.1.1 Service Layers

Services have been divided into six logical categories: Presentation, Business Process, Business Activity, Data, Connectivity, and Utility Services representing Services generally derrived from the SOA infrastructure. The sections that follow discuss the concepts and capabilities of each Service type.

Figure 2–2 Service Layers

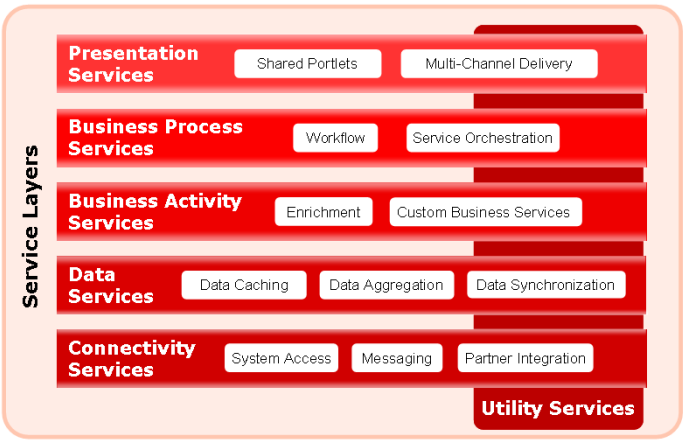


Figure 2–2 above breaks out the six service layers from Figure 2–1, "SOA Logical Model"

2.1.1.1 Utility Services

Utility Services is a classification of Services that are not directly related to performing business operations. That is, any Service that is not providing the connectivity, data management, business, or presentation logic associated with a business activity. Utility

Services generally perform infrastructure-related functions, such as security (credential mapping, access control, auditing), logging, notification, policy lookup, transaction watermarking, etc. This classification of IT functions is often referred to as "cross-cutting" and are, therefore drawn perpendicular to the other services in this diagram; they also typically rely heavily on the SOA Infrastructure (see below) for their implementation.

Utility Services do not carry any business context or have an affinity to any specific application; they are often the simplest and most reused Services in a SOA environment.

Architectural Principles

1. Utility Services must not contain business logic

2.1.1.2 Connectivity Services

Connectivity Services provide a means to bridge current application technologies with SOA. They establish connectivity with legacy applications that do not inherently provide Service oriented interfaces. Connectivity may be accomplished using a variety of means, such as messaging systems, application adapters, and custom code. Connectivity Services expose the functions of legacy systems as Services in an SOA environment. They provide standards based interfaces, such as SOAP, which are then consumed by higher level Services or client applications through Service infrastructure.

Services exposed at this layer are not intended to reflect business context in any way. Being context-neutral, they can be used in many different business scenarios. Data obtained by these Services will usually map directly to data representations in the native legacy systems. Adding business context or data mapping will effectively promote them to a higher level (business or data Services).

Architectural Principles

1. Connectivity Services must be stateless.
2. Connectivity Services may translate the representation of data, e.g., to and from XML, but do not apply rules-based transformation or aggregate data from multiple sources.
3. Connectivity Services must not include business logic.

2.1.1.3 Data Services

Data Services are used to access data from various sources using many different technologies, and present data in a business-friendly form. Data may originate in various databases, flat files, XML files, and legacy systems. Data Services offer a way to aggregate, transform, and synchronize data from multiple sources. They expose data in a format that best supports business Services and composite business applications. This creates an abstraction between the users of data and the sources of data. Users of data do not need to be concerned with where data is stored, how it is stored, or how it is represented in its native form. They can work with representations (schemas) of greater business value, such as canonical data models.

Data Services may access data directly, or may use Connectivity Services to obtain data from legacy applications. Connectivity Services provide a means to interact with legacy systems through APIs rather than directly to the database. Though this may add processing overhead, it is often preferred when updates to data might affect applications that rely on the data.

Like Connectivity Services, Data Services do not include any business logic. This allows them to be reused in different business contexts without modification. They represent business objects, such as customer, employee, and order, with the associated logic for ensuring consistency of such constructs.

Modern development tools can improve the development of data Services. They provide visual editing and mapping capabilities that allow aggregation and transformation Services to be configured without cumbersome low level coding. They can also provide enhanced features, such as caching and redaction. Redaction is the ability to reduce the data returned by a Service according to the entitlements of the user invoking the Service. This permits the same Service to be used by different users in different roles without having to hard code change to results sets.

Architectural Principles

1. Data Services must be tied to an enterprise information model
2. Data Services must not contain business logic
3. Data Services must be stateless
4. Business Services drive the need for data Services
5. Data is owned by the enterprise - accountability and stewardship must be delegated to the authoritative business source
6. Data Services should enforce the separation of data access from the use of data by other Services i.e. other Services should not perform their own data access

2.1.1.4 Business Services

There are many similarities between Business Activity Services and Business Process Services, shown in the diagram in Figure 2–2, so this section describes the common characteristics first, followed separate descriptions detailing their differences. The sometines subtle reason for the separation of the two types of busines services is that, while Business Activity Services focus on static business functions, Business Process Services provide dynamic workflow orchestations of business functions.

Business Services are the most commonly talked about Services in a SOA. They expose business operations via Service interfaces to establish a single, rationalized way to perform business functions. Operations may be simple atomic business functions, such as obtaining a stock quote, or complex / long running, such as starting a workflow process or kicking off a batch process.

Business Services represent operations that either have a high likelihood of reuse, or a significant value to the organization. Engineering and exposing these operations as Business Services promotes interoperability between systems and creates a discoverable, published, rationalized, and managed way of performing these functions. It also promotes agility if the Services are packaged and managed as separate deployable and executable units. Services can be modified far more quickly and easily as standalone modular units than operations embedded and intertwined in large complex applications.

The reuse potential of Business Services is generally less than Data and Connectivity Services. As the level of complexity increases, the likelihood of reuse is reduced. For this reason Business Services offer more value as a way to rationalize IT operations and achieve agility than as a way to promote reuse. By abstracting the users of Services from the system interactions needed to perform those Services, they provide a way for IT to evolve its systems without a direct impact to the users.

Simple Services may be developed using ordinary low level development platforms, such as J2EE or .NET, while more complex Services might be created with workflow

management tools. They may be stateless operations that involve a single atomic transaction, or stateful processes that interact with multiple applications and Services asynchronously.

Architectural Principles

1. All transactional processing required by a Business Service must be self-contained within the Service. Transactions cannot span Service boundaries.
2. Business Services should be individually packaged and deployed.
3. Business Services are presentation-neutral, e.g., do not include formatting and presentation metadata.
4. Business logic must be separated from underlying technology and delivery channels.

2.1.1.4.1 Business Activity Services are typically identified through a top-down method of functional decomposition, they are units of business logic offered as Services. The business logic may be simple operations, such as calculating sales tax, or more complex operations, such as transferring money electronically in a banking system.

This category of Services may also include business rules functionality (Services offering Business rules are often called Decision Services).

Architectural Principles

1. Business Activity Services must be stateless.
2. Business Activity Services should be designed to be idempotent (i.e. perform their action once regardless of the number of times called; this avoids duplication when an action is retried for any reason).

2.1.1.4.2 Business Process Services are typically identified through a top-down method of process decomposition and represent shared business processes. They can be complete business processes or independently defined subprocesses. For example, the process for fulfilling a product order may differ slightly depending on the manner in which it was ordered. The common portion of all order processes may be defined once and reused by other higher level processes that are unique to the method of ordering. This level of reuse is most valuable for establishing consistency for doing business. Exposed as Services, these subprocesses can be easily incorporated into higher level processes while being managed by Service infrastructure.

Business Process Services are units of business logic just like Business Activity Services, but the logic is likely to represent more complex operations such as orchestrating the interaction with multiple systems to handle a customer service enquiry.

The, potentially complex, process orchestrations of Business Process Services are likely to involve business rules to support process branching and compensating transactions to support fault handling in business processes.

Architectural Principles

1. Business Services should have an "undo" method to correspond with each "execute" method (see "compensating transations" below).

Compensating Transactions: unlike simple database "undos", long running business processes cannot be simply undone. In cases where a series of activities are preformed, in the execution of a long running business process, the reversal of the process (in the event of error condition, or a processing exception for higher-level business reason) may not as simple as undoing the individual activities that make-up the process. An

example of a situation requiring a compensating transaction might be a process committing resources in anticipation of user input; if the input is not received within a predetermined time the process fails, however the resource cannot be simply replaced and some form of compensating action must be performed.

2.1.1.5 Presentation Services

As in most application architectures, the highest layer involves interaction with clients. If a client happens to be an end user, then the information provided by the system must be merged with metadata to describe how that information is formatted and presented. Since all Service layers discussed so far are presentation-neutral, the consumer of those Services must perform presentation-specific formatting before the information reaches the client.

In addition to formatting, a User Interface (UI) layer will often merge data from multiple unrelated sources in order to create the most beneficial experience. For example, if a user is browsing a parts catalog, the system may include a selection of tools for sale. The proper place to merge this data is at the presentation layer, since there isn't any structured relationship between parts and tools.

There are occasions where reuse can be beneficial after presentation formatting and/or information aggregation is performed. The most common example is a portal. The portal application is divided into portlets which contain information from various sources and the ability to interact with underlying systems. Using portal federation technologies such as WSRP, portal applications can be developed and deployed in a componentized manner. That is, the portal framework (layout, look and feel, security) can be developed separately from the portlets it displays. Furthermore, portlets may be developed and deployed separately from each other. This permits a separation of concerns with regards to system development and ownership. The groups responsible for Services and applications that the portlets expose can develop and maintain the portlets themselves. Portal applications can then consume these portlets as desired.

Portlets can be deployed as part of the shared Services infrastructure and leveraged by many different portal applications. Their greatest value is the ability to reuse entire functional capabilities including presentation logic. Reuse at this level avoids the need for re-coding presentation logic. Like Business Services, Presentation Services may represent simple operations (stock quote, calculator), or complex processes (order management, customer service).

Presentation Services are defined as Services whose response includes metadata to describe how information is presented to the user. Besides portlets, any Services that return data in the form of HTML or XHTML would qualify. The key benefits of Presentation Services are: (i) reuse of presentation formatting logic, and (ii) the ability to provide content to multiple channels via a single Service, i.e., support for multi-channel delivery.

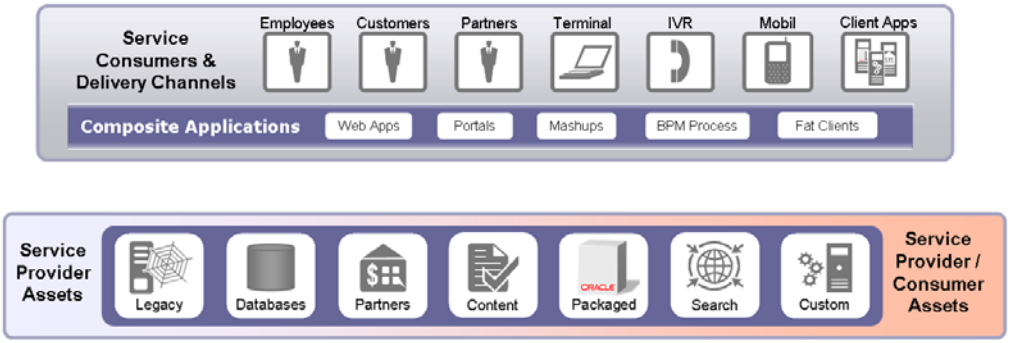
Architectural Principles

1. The presentation layer must separate data (information) from its presentation (format, layout, etc.)
2. Presentation tier will be completely devoid of any workflow, business process, business rules, etc.

2.1.2 Service Providers and Service Consumers

Figure 2–3 shows "Service Consumers" from the top portion of Figure 2–1, "SOA Logical Model", while the combination of "Service Providers" and "Service Consumers" is taken from the bottom group of elemnts.

Figure 2–3 Service Providers and Service Consumers



The distinction between Service Providers and Consumers, their different mainfestations, and the reason for this distribution in the diagram is explained in the following sections.

2.1.2.1 Service Providers

Service providers come in many forms, including legacy systems, packaged applications, partner applications, messaging systems, custom-built standalone Services, business processes, and combinations of these.

It is important to note that providing a Service is more than simply providing an interface to invoke a business function. The Service provider must ensure that qualities of Service are stated, offered, and adhered to. This is especially important for Services of an enterprise-class nature, i.e., Services advertised for use across departmental boundaries and/or Services that are leveraged by mission-critical solutions. Therefore the principles of Service enablement must be followed.

Architectural Principles

1. An interface must be provided that conforms to the reference architecture
2. A contract must be provided as specified in the reference architecture
3. Security policy must be defined and enforced
4. Versioning strategy must be adhered to
5. The Service must conform to infrastructure rules for discovery, management, monitoring, and governance
6. The Service must be classified according to layers and definitions of the reference architecture
7. The Service provider must ensure that the Service will satisfy the aggregate specifications of all related usage agreements

2.1.2.2 Service Consumers

Service consumers are, of course, consumers of Services, but more importantly they do not offer an 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.

2.1.2.3 Composite Applications

Composite Applications is a term used to define applications built primarily of Services. These Services may belong to any of the Service layers previously described. The applications may dictate the need for Services, identifying business needs that can be developed in the form of Services. This is often referred to as top-down Service identification. In essence, the application is defined first, and the Services used to compose the application are identified and constructed. Composite application may evolve and expand over time as additional Services are created. This supports incremental business value through the iterative development of new Services, reducing the risk associated with applications by delivering functionality in smaller, more consistent increments.

As the portfolio of Services increases, the ability to rapidly compose new applications increases. The ratio of Services that already exist vs. Services that need to be built changes over time. Eventually, composite applications may be composed almost entirely of existing Services. At this point it is conceivable that applications can be built without any development effort at all.

Common examples of Composite Applications include portals and BPM processes. Portals may be composed of Presentation Services, while BPM processes may be composed of Business, Data, and Connectivity Services. Technologies are evolving in this area to allow these types of applications to be configured rather than developed. This will allow more to be done by business analysts and end users, further reducing development efforts required to meet application needs and speed time to market.

Composite Applications consume Services, they are not service providers since no services are exposed and therefore, they cannot be classified as a Service themselves; however they may have "Servicable" functionality, but this is deployed separately. As Service consumers, Composite Applications are positioned at the top of the diagram in Figure 2–3.

The BPM Process contained in Composite Applications in Figure 2–3 refers to any type of process that is a service consumer, but not a service provider; other processes exposed as services would all be classified in the Business Process Service layer.

2.1.2.4 A note about Composite Services

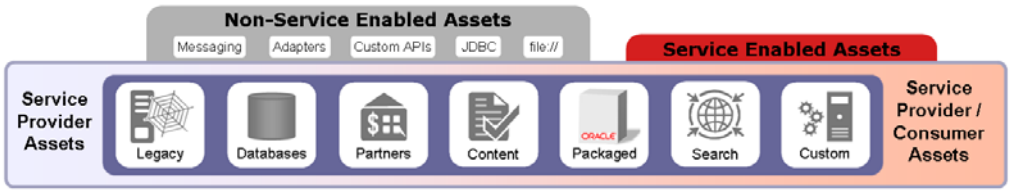
Unlike Composite Applications, Composite Services do not provide complete, self-contained functionality to an end-user, but instead provide Services to other Services, systems, or applications. Composite Services are comprised of other Services and are hence, Service Consumers; since they provide Services they are also Service Providers.

More than simply an aggregation of Servcies, Service Composites must comply with the full definition of a Service in the true sense (encompassing governance, architecture compliance, discovery, and reuse) and require the discipline of classification. Service Composites have no need for a separate service layer or classification of their own, instead they are classified in the diagram in Figure 2–1 based on the highest architecture layer they exhibit.

2.1.2.5 Service Enabled v. non-Service Enabled Assets

Figure 2–4 below shows IT assets from the bottom portion of Figure 2–1, "SOA Logical Model" and the distinction between Service enabled and non-Service enabled.

Figure 2–4 Service Enabled v. non-Service Enabled Assets



Service enabled assets, sometimes referred to as "SOA-enabled applications", may be a service consumers, service providers, or both. Service enabled applications that are service providers are essentially applications deployed with embedded services; they can be subcategorized into various types such as Packaged Apps, Partner Apps (which includes **SaaS**, **cloud**, etc.), and Custom Applications. Service enabled applications that are service providers, whether or not they are also service consumers, are depicted at the bottom of the diagram in Figure 2–4.

Service enabled applications that are only service consumers are referred to as "client applications". Client applications differ from composite applications in that they are not originally designed for SOA. Typically client applications consume services merely for the benefit of the integration capabilities of available Services. On the maturity scale client applications are 'opportunistic' at best. Client appliations represent the cases where traditional applications happen to invoke services. As service consumers, they are depicted at the top of the diagram in Figure 2–4.

Note 1: Not all packaged or partner applications are Service enabled.

Note 2: Custom applications are Service enabled, otherwise they are considered legacy applications.

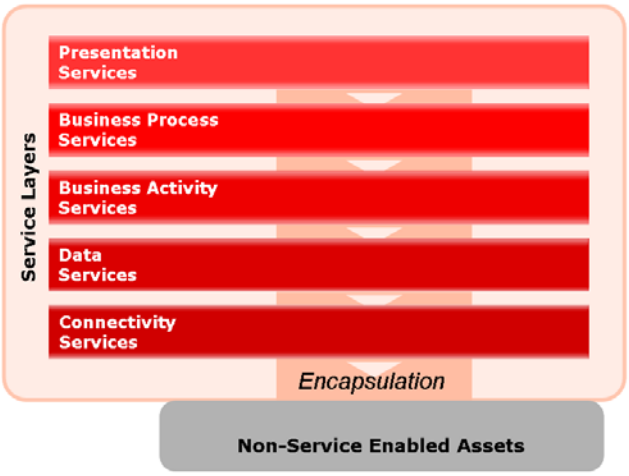
In contrast, non-service enabled assets (also referred to as Legacy or Traditional applications) do not consume or provide services: instead their functionality (or data) is merely wrapped (or encapsulated) and used by other services either via the connectivity Serice layer or embedded directly within a Service of another layer.

2.1.2.6 Encapsulation v. Service Enablement

As we can see from the diagram in Figure 2–4 the collection of legacy, packaged, custom application, etc. in the bottom box makes up a wide spectrum of Service providers and consumers with the only clear separation being Service enablement (yielding Service enabled assets) v. encapsulation of non-Service enabled assets.

Service enablement, in this specific case, refers to the process of extending an IT asset, using SOA infrastructure, to add the necessary capabilities to ensure that it conforms to the SOA definition.

Figure 2–5 Encapsulation



In contrast, encapsulation refers to inclusion of a function from lower layers or simply wrapping a legacy asset in order to use its functionality inside another Service. Any service layer can include legacy functions using a suitable wrapper; for instance, a business activity service could enrich a legacy application and expose this as a service without the use of a connectivity service; similarly a presentation service could include business logic, data aggregation, and legacy system connectivity. The decision whether to encapsulate everything in a single service or split it into multiple services is a function of service justification during analysis.

2.1.3 SOA Infrastructure

The SOA infrastructure plays a major role in the realization of many of the non-functional capabilities of a service.

SOA infrastructure is a major topic in its own right and is therefore covered in a separate document *ORA SOA Infrastructure*.

2.2 Types of Services and Service Scope

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

- Multi-Enterprise
- Enterprise Wide
- Intra-Line of Business (LOB)
- Intra-Application

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.

2.2.1 Multi-Enterprise or Public Services

Multi-enterprise 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.

2.2.2 Enterprise-Wide

Enterprise- wide Services are available to qualified consumers throughout the enterprise. These Services are not exposed to consumers outside the enterprise.

2.2.3 Intra-Line of Business (LoB)

Intra-LOB Services are those that 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.

2.2.4 Intra-Application

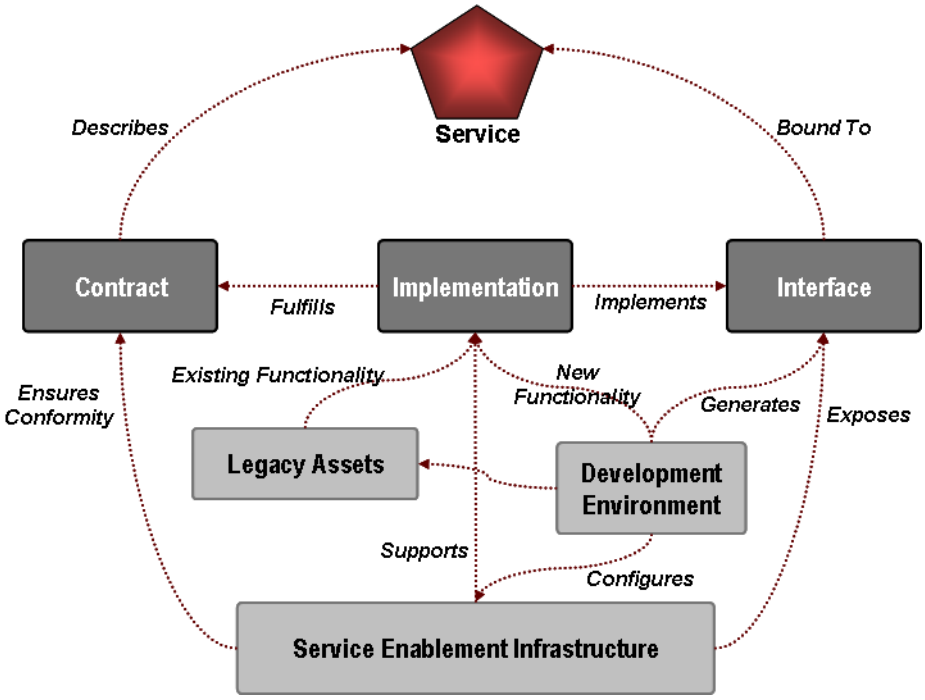
Intra-Application Services are those Services that are not intended to be called outside of the scope of a given application. Typically these Services are not hosted in a shared Service environment but with the application itself. Quite often these Services result from the failed justification of a Service candidate for a wider scoped Service.

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 2–6.

Figure 2–6 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 2–7 Service Contract Template

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.

2.3.2 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 process-able 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 Technology 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 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 process-able 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.

2.3.2.1 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.

2.3.2.2 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

2.3.2.3 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.

2.3.2.4 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.

2.3.3 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.

2.3.3.1 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-so 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.

2.3.3.2 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.

2.3.4 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 Simple UML Profile for Services

The Service meta-model shown in Figure 2–6 provides a visual depiction of the aspects of a Service. It does not, however, provide the rigour of a UML profile that would be suitable for modeling Services via UML.

Appendix A describes a Service in terms of a UML profile, based on the components of a Service that were defined and described above. The UML profile is actually a method for modeling Services, i.e. a meta-model. It can be used to provide direction on how to model Services and their constituent parts.

2.5 Separation of Concerns

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. The objective of this section is to describe architectural principles that enforce this separation of concerns to help maximize versatility and reuse.

Services should be written to accomplish their function regardless of what protocol is used to invoke them, where they physically exist, or on what type of hardware or operating system they run on. This provides for maximum reuse by allowing access through multiple types of interfaces. It also provides greater versatility in how they are deployed and what underlying technologies are used.

Architectural Principles

1. A Service must not have any dependency on the identity of the consumer that invokes it.
2. Services must not be tied to any particular underlying technology, delivery channel, or physical location.

2.6 Service Versioning

Services, like any other code assets, 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, such as when an underlying database change is made. 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.

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

Technologies and Standards

SOA must be standards-based in order to achieve interoperability. Also for given Enterprise, technologies (in the form of platforms), tools, and engineering practices must adhere to selected standards.

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.

3.1 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

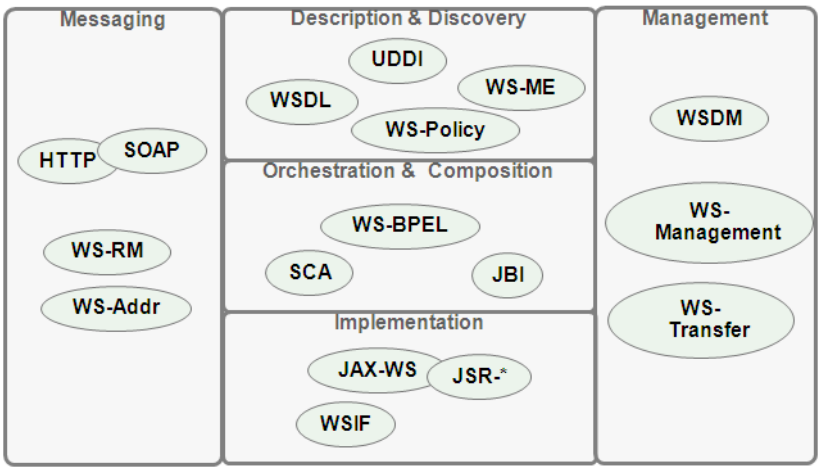
- HL7 (XML messaging standard for Healthcare for the exchange, integration, sharing and retrieval of electronic health information)
- HRXML (a library of XML schemas developed by the HR-XML Consortium, Inc. to support a variety of business processes related to human resource management)
- Telecoms IMS (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 3–1 Categorization of SOA for standards



The categories and their core standards shown in Figure 3–1 are described in following sections.

3.1.1 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 & 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 & 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), 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 & WSDL typed variables, XPath for expressions & 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.

3.1.1.1 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

3.1.2 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.

3.1.3 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 3.2.4, "Web Services Standards" of this chapter.

3.1.4 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 & 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 & deployment) (JSR181)
- SOAP with Attachments API for Java (SAAJ) (JSR67)
- WS for Java EE 1.2 (WSEE) (JSR109)
- JAX-R interface fro UDDI repositories (inc. with J2EE 1.4 & Java EE 5)

3.2 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.

3.2.1 REST

The unprecedented success of the Internet is largely attributable to Internet Protocol (IP) and within that Hypertext Transfer Protocol (HTTP), the commonest 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 3.2.3 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.

3.2.2 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.

3.2.2.1 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 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".

3.2.3 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

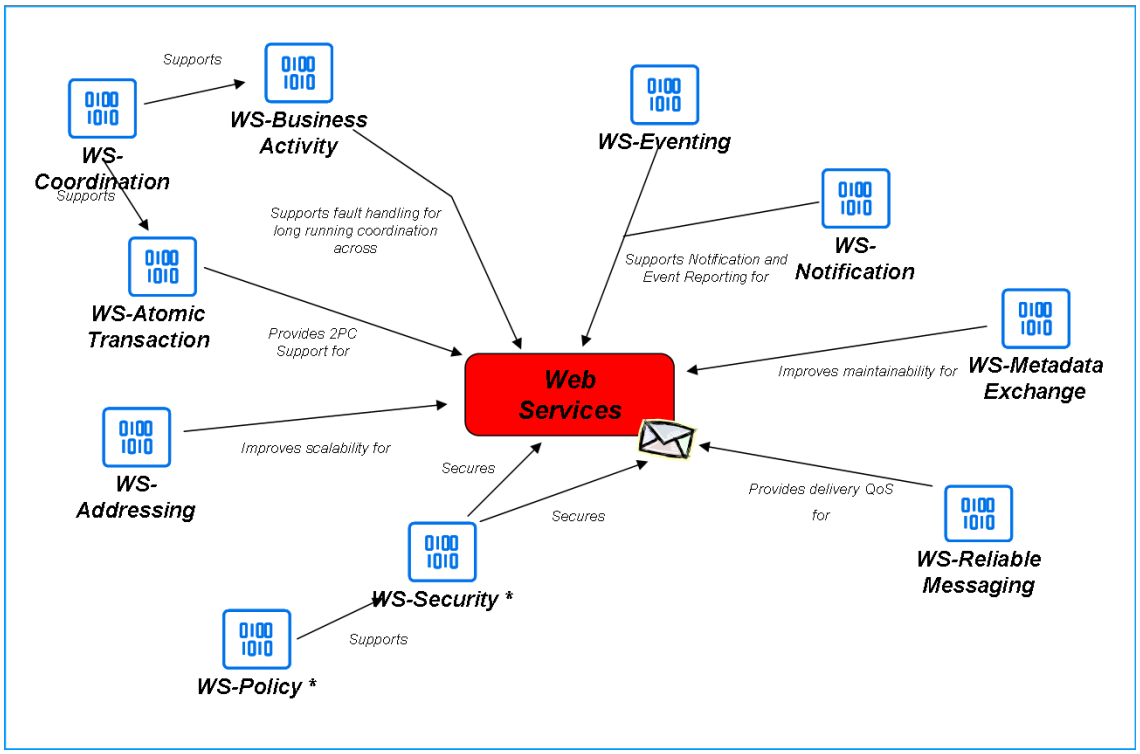
One of the major inhibitors of SOA adoption, at least in the beginning, was the ability to achieve certain levels of QoS through a Service Oriented Architecture. The initial release of specifications utilizing Services in the form of Web Services had very weak support for various standard levels of QoS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

3.2.4 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 exhustive 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 3–2 Web Services Specifications



The following sections provide an outline of the standards specifications in Figure 3–2 above.

3.2.4.1 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)

3.2.4.1.1 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.

3.2.4.1.2 WS-AtomicTransactions 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.

3.2.4.1.3 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.

3.2.4.2 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.

3.2.4.3 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).

3.2.4.4 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.

3.2.4.5 WS-Reliable Messaging

This specification is designed to support the delivery based quality of Services for messages (eg: guaranteed delivery, at most once, etc). Partial support for this specification is provided by most Service infrastructure vendors.

3.2.4.6 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.

3.2.4.7 WS-Eventing & 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.

3.2.4.8 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 however, instead it creates guidelines and tests for their interoperability.

3.3 Infrastructure

This section describes the capabilities that should be expected from a Service infrastructure. Descriptions of infrastructure implementation and product mapping are not included here, but are covered instead by the *ORA SOA Infrastructure* document.

3.3.1 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. Service enablement is responsible for the following aspects of a Service:

- Configuration based logic
  - Routing
  - Transformations
  - Versioning Support
- Composition
  - The ability to compose a new Service through the composition of two or more Services.
- Change Control
  - Audit support
  - Configuration aggregation
  - Configuration Rollback support

  - Real-change application. Changes are applied without requiring a redeployment or server restart.
- Throttling
  - Thread management
  - Priority based resource management
  - Consumption monitoring
- Security
  - Centralized Policy Management
  - Distributed Enforcement
  - Security Standards (WS-Security, WS-Policy, SAML, SSL, etc).
- Interface exposure
  - Using accepted interface protocols, transports, message types, and standards
  - Service Discovery (published to a registry and/or repository)
  - Service Definition (creation of a contract)

As far as possible these Service enablement capabilities should conform to SOA standards. Due to the fact that Service enablement is provided via infrastructure, rather than code, its has a significant impact on Service deployment (see Chapter 4, "Packaging and Deployment" for more information).

The following infrastructure capabilities may be utilized at the Design Time to reduce some of the complexities of managing and deploying SOAs of any shape, form and maturity as well as enforce polices and procedures established by an organization around SOA:

- Centralized Service Information Repository
- Service Discovery Capabilities
- Service Version Management
- Service Composition Design
- Security Policy Management
- QOS Design Capabilities (SLA and Exception Management)
- Environment Propagation Capabilities

Keep in mind, that some of the above capabilities have direct impact on the requirements for the Runtime infrastructure selected. Further, the order in which these capabilities are required by an enterprise adopting SOA varies. Functionally, the capabilities listed above can be accomplished with the design time components of the logical architecture. These components consist of the following:

- Enterprise Repository
- Application & Service Design/Composition
- Service Management
- Security Management

3.3.2 Service Discovery

There are two sides to consider regarding Service discovery: design time discovery and run time discovery. Design time discovery pertains to the way an architect or application designer finds out that a Service exists. There must be some form of capability to look for Services, such as a keyword search mechanism or a navigable taxonomy. Once the Service is found, there must be sufficient information about the Service to determine if the operations, data inputs and outputs, response time and load capabilities, transactional support, security, etc. meet the potential consumer's needs. This design time discovery is discussed further in the *ORA SOA Infrastructure* document.

Run time Service discovery pertains to the process of locating a physical Service provider and Service endpoint when the Service is invoked. In the traditional application integration approach, the consumer and provider are hard wired together. That is, the consumer knows where the provider is located, e.g., the network address where the interface is handled. If the address changes or the interface is modified by the provider, then the consumer must change in lock step. Otherwise the consumer (all consumers) will fail until appropriate changes are deployed. In order to avoid such a situation, SOA usually involves some form of run time discovery of Service endpoints. This can be handled in a number of ways, which are discussed in the *ORFA SOA Infrastructure* document. What is important to mention here is that SOA infrastructure provides loose coupling between consumers and providers. This helps avoid the problems associated with hard-wired interfaces, which promotes interoperability and agility.

3.3.3 Service Mediation

In addition to run time Service endpoint discovery, SOA infrastructure can provide additional value by acting as an intermediary and mediator between consumers and providers. For example, intermediaries can bridge the technology gaps between the two parties. Among their many capabilities are:

- Translate, or transform request and response messages
- Accept requests via one transport or protocol and forward them on using a different transport or protocol
- Route messages based on content within the request message (Content-based routing)
- Route messages based on security policies
- Translate (map) security credentials between different users/groups/roles or between different credential types
- Add or remove security measures such as encryption and certificates
- Invoke multiple Service providers as part of a single Service request
- Audit and/or log requests
- Deny requests based on access policies (SLAs, Usage Agreements)
- Capture response time metrics and usage metrics
- Monitor and report on error conditions

Architectural Principles

1. Services must be loosely coupled, e.g. they must always be accessed via an intermediary. Consumers must not directly access Service provider endpoints.

This topic is covered in more detail in the *ORA SOA Infrastructure* document.

3.3.4 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), or via configuration data (config-based routing).

Loose coupling is the ability to decouple 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. Routing capability enables the loose coupling required for SOA. These features are fundamental to managing and maintaining a mature SOA deployment.

3.3.5 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, which then accesses the appropriate endpoint.

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

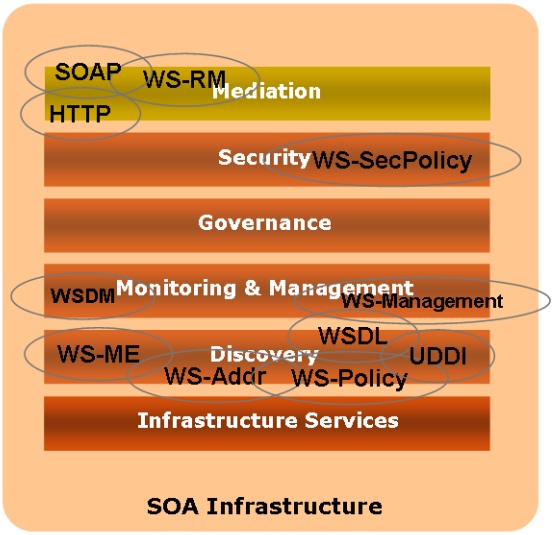
3.3.6 Error Handling

Error handling capabilities allow 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 recovery from the error.

3.3.7 Infrastructure Standards

Looking back at the earlier standards discussion we can now map the relevant standards to the infrastructure. The following diagram takes the SOA infrastructure portion of the Logical Architecture (described in Chapter 2, "Architectural Models") and maps the relevant standards to the appropriate capabilities.

Figure 3–3 SOA Standards Mapped to 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 here in Figure 3–3, such as XML Digital Signatures associated with WS-SecurityPolicy; many others are not shown because they are not exclusive to the SOA domain.

4

Packaging and Deployment

As discussed earlier in Chapter 3, "Technologies and Standards", Service deployment (and hence its corresponding packaging) is a broader subject than traditional application deployment due to the diversity of SOA infrastructure. This chapter explores the new demands of both packaging and deployment for SOA.

4.1 Packaging

Packaging of software applications for deployment purposes is an onerous task, due mainly to great number of moving parts required to be synchronized to create a complete functional unit. Component based architectures, such as those supported by Java Enterprise Edition containers, now have rich tooling options to help support the packaging of a plethora of configuration files (descriptors, directories, etc.) along with the code itself (into "archive" files, JAR, WAR, EAR).

The key resources that may be modified at deployment time are:

- Java EE Server Resources
  - Clustering
  - Java Message Service
  - EJB Pooling and Caching
  - JDBC Connection Pools
  - Execution Thread Pool
  - Java EE Connector Architecture

Under the approach of Service enablement, in which existing application functionality is exposed as a Service through the SOA infrastructure capabilities (rather than new code construction), a Service is potentially deployed without complexity of traditional application deployment. This new Service infrastructure however, brings its own complexities to the construction of deployable packages of Service capabilities.

Common types of deployable artifacts in a SOA environment generally include the following:

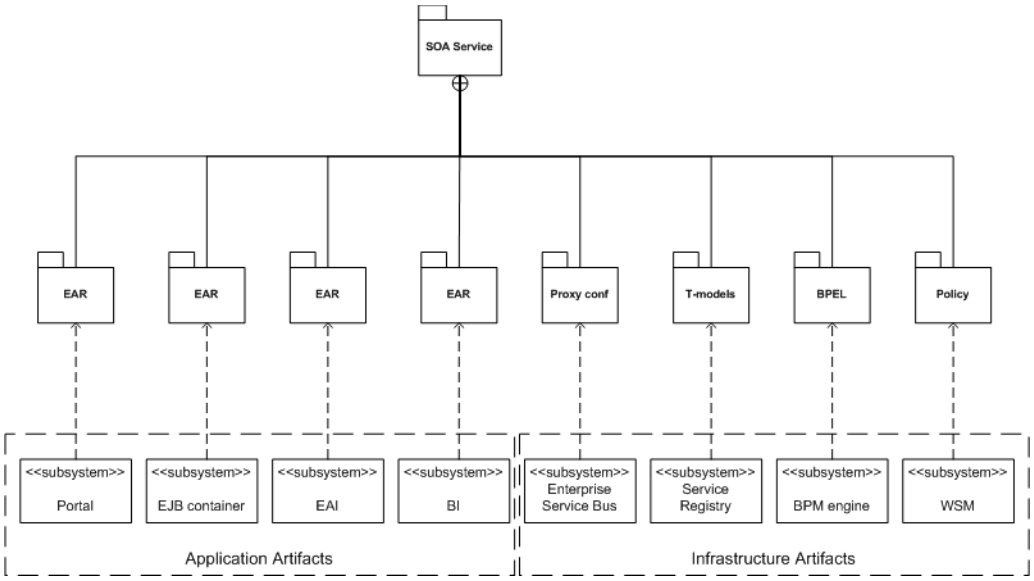
- WSDL, XML Schema, XSLT, Xpath, various rules definitions, WS-* docs
- Code (in the form of JAR, WAR, EAR)
- Directory definitions
- Access controls

The approach to Service packaging is dependent upon the infrastructure being deployed to support it. The following list of topics breaks down the packaging artifacts by its infrastructure type.

- Service Bus Configuration Resources
  - Business Service Endpoint definitions (URIs)
  - Proxy (intermediary) Services
    - Message routing, directory services, environmental information
  - Message Format Language (MFL) is a language used to define rules to transform formatted binary data into XML data
  - WSDL for interface and endpoint definitions
    - XML Schema for type definitions
  - Xpath and XSLT for transformations
  - WS-Policy, Addressing, ReliableMessaging, etc. docs for QoS & access control
  - JARs to store compiled Java classes and associated metadata that constitute a program code
  - Other config: Alert destinations, JNDI, UDDI, SMTP servers, etc.
- Service Registry
  - metadata for various types of software asset
    - business processes
    - Services
    - Patterns
    - Frameworks
    - Applications
    - Components
  - Contract, Policy, and interface definitions
    - WSDL
    - WS-Policy
    - Many other WS-* documents
- Other infrastructure resources potentially requiring packaged configuration
  - Web Services Management
  - Business Process Management
  - Rules Engine
  - Master Data Management
  - Relational Database System Resources
  - Hardware, Operating System, and Network Resources

These artifacts of Service deployment still be depicted using traditional software diagramming techniques, such as UML. An example high-level packaging model is shown below in Figure 4–1 to represent the mapping of deployable Service implementation artifacts to their respective infrastructure sub-system components, determined by the needs of a specific Deployment Architecture.

Figure 4–1 Example High-level Packaging Diagram

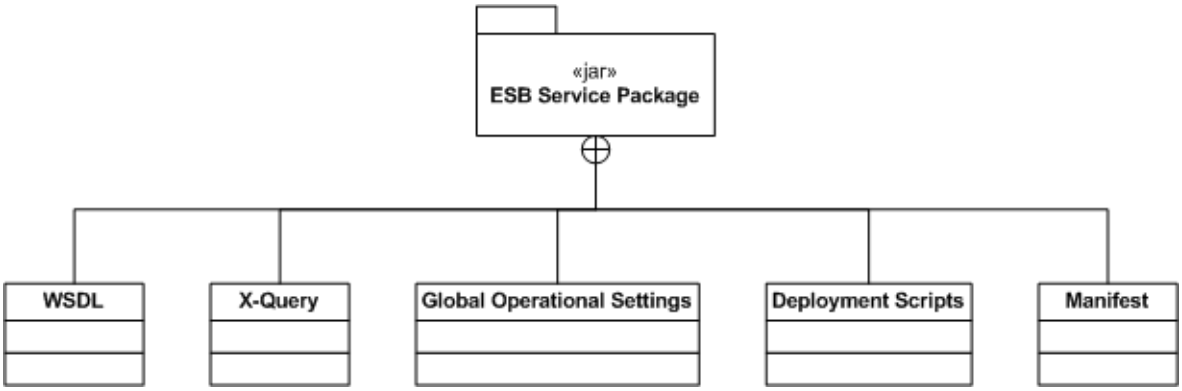


Representing packages and their infrastructure sub-systems can serve two important purposes:

- Packages of Service Implementation artifacts should be (as far as reasonably possible) self-contained to simplify the process of Service deployment
- Traceability of packaged Service Implementation facets to sub-systems in the runtime system.

Each sub-package in the high-level model below requires substantially more detail than is shown here. More detailed models can be used to show the types of artifacts that can be expected in each of the high-level sub-packages. An example of a of a more detailed package diagram is shown below in Figure 4–2.

Figure 4–2 Example Service Packaging Detail Diagram



4.2 Deployment

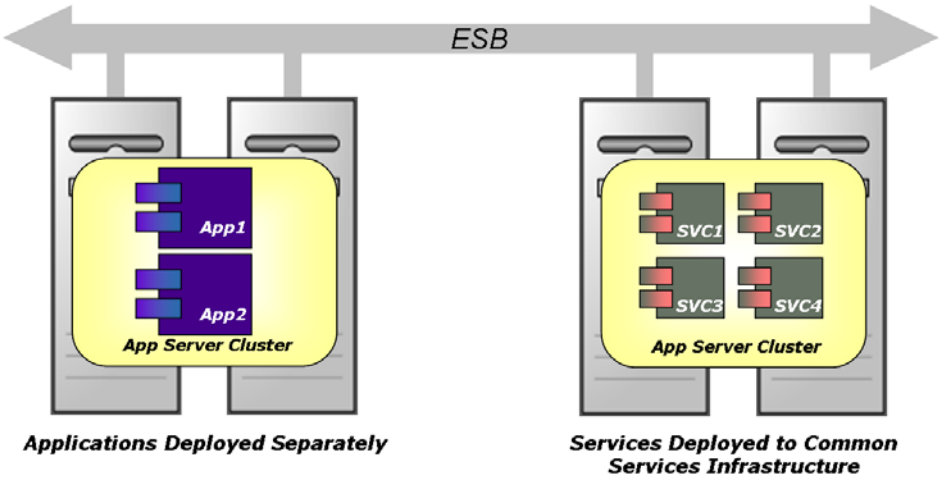
For the purpose of this document, deployment is approached from a logical view rather than a physical view. The logical characteristics include the strategy for Service deployment and the federation of Services and infrastructure. A physical view of Service deployment can be found in the *ORA SOA Infrastructure* document.

The key aspect of Services deployment is the ability to ensure the environment can support the needs of the Services. For sharing of assets to work, users must be confident in their reliability, availability, and performance. While reliability is a product of software quality, availability and performance are generally dictated by the operating environment.

4.2.1 Segregated Deployment

The approach, recommended by Oracle, to Services deployment is the separation of Services from applications and the grouping of Services with similar availability needs. To accomplish this, projects must be logically divided into shared and non-shared parts. The shared parts become Services, while the non-shared parts become ordinary applications. Development of these assets can be done by the same team; however it is important that they are constructed in a way that allows them to be deployed separately, as illustrated below in Figure 4–3.

Figure 4–3 Segregated Deployment



This illustration shows an application cluster used to host ordinary applications, and a Services cluster hosting Services. The clusters connect through the Services bus, allowing access to Services by the applications. Redundant hardware is used in both clusters to provide high availability.

The Services cluster can contain Services of many types, e.g. Connectivity, Data, Business, and Presentation Services. Mixing these in a cluster is not an issue. The important considerations for grouping Services are:

- Ownership of resources. This includes hardware and software ownership, and responsibility for management. At a high level we would expect assets managed at the city level to be grouped separately from those managed at the province level. Further sub-grouping of assets would depend on the organizational structure of each location.

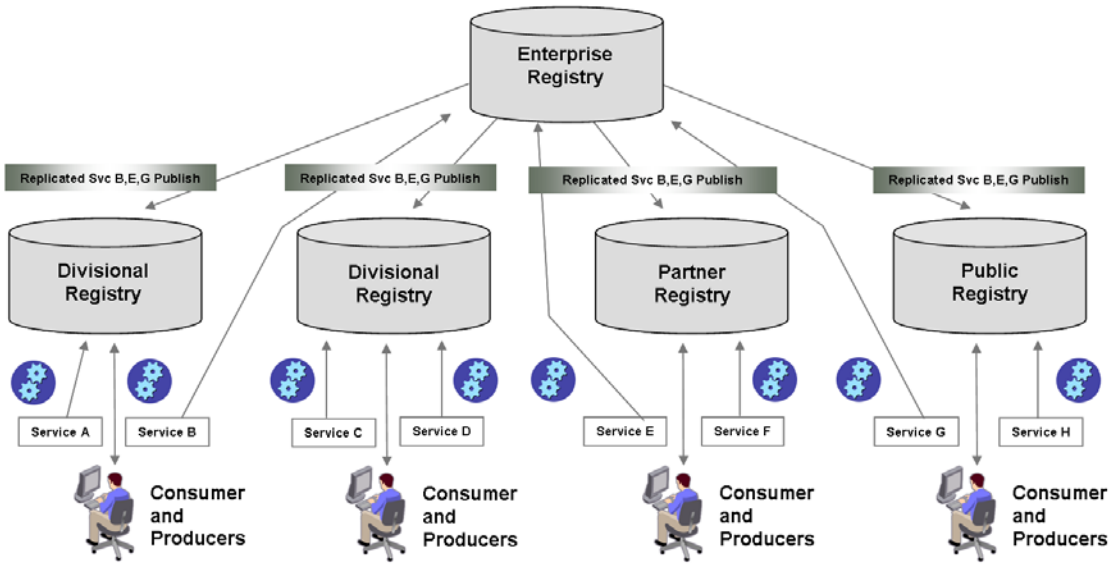
- Technology. Different technologies used for Service implementation will require different clustering strategies and may require different hardware platforms.
- Service Level Agreements. Services that demand high availability and low downtime or maintenance windows should be separated from Services that do not share these requirements.

4.2.2 Federated Deployment

In a large, geographically distributed Enterprise, deployment of Services and infrastructure must support a federated model. This permits a level of autonomy for each organization and at the same time promotes Service reuse, cooperation, coordination, and information sharing.

Under such a federated model each organization (division or LoB) will have the ability to host its own Services and infrastructure. This grouping of assets can be referred to as a domain. Each domain will be able to host its own Services and determine which Services should be offered to other domains. The discovery of Services in other domains is accomplished via a federated registry, as illustrated below in Figure 4–4.

Figure 4–4 Federated Registry



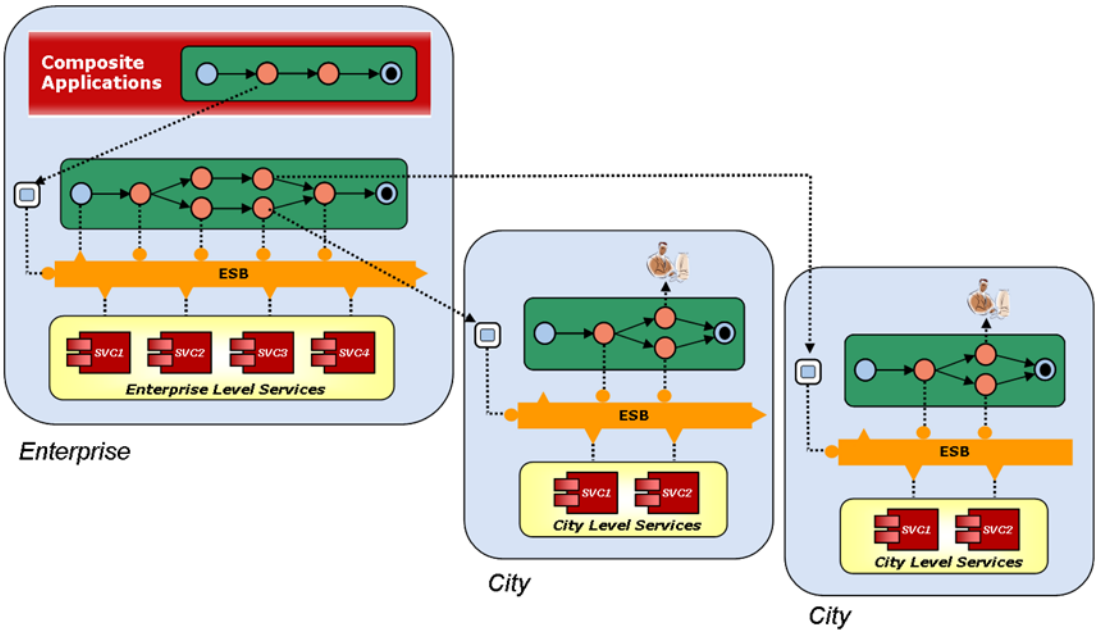
Federation will allow developers to discover Services in other domains without having to navigate their web browser from one domain to the other. It will also allow Services to be organized in an enterprise taxonomy as opposed to multiple unique domain taxonomies. While the public registry should be federated, the local domain should not.

Deployment options for federated registries will depend on the actual product that is chosen. If the registry is only used for discovery (i.e., dynamic binding is not permitted) then high availability will not be such an issue.

The Service bus will also be deployed in a federated model - each domain with its own bus. Local Services will only have a proxy on the local bus. Enterprise Services may be proxied in two different ways. The first way is to only provide a proxy on the local bus. Service consumers from other domains would access the remote bus directly. This is the simplest to manage since only one proxy is needed and both registries (private

and public) can have the same Service endpoint configuration. Another way is to have all Service consumers connectivity Services via their local bus. It requires the local bus to proxy to the remote bus, which then proxies to the actual Service. Management is more difficult since proxies for public Services must be configured on each Service bus in the entire enterprise. This method may be considered if security considerations necessitate a secure channel between domains. For instance, in the case of highly secure data transfer, an encryption and decryption method may be established between domains, which is facilitated by the bus on each end.

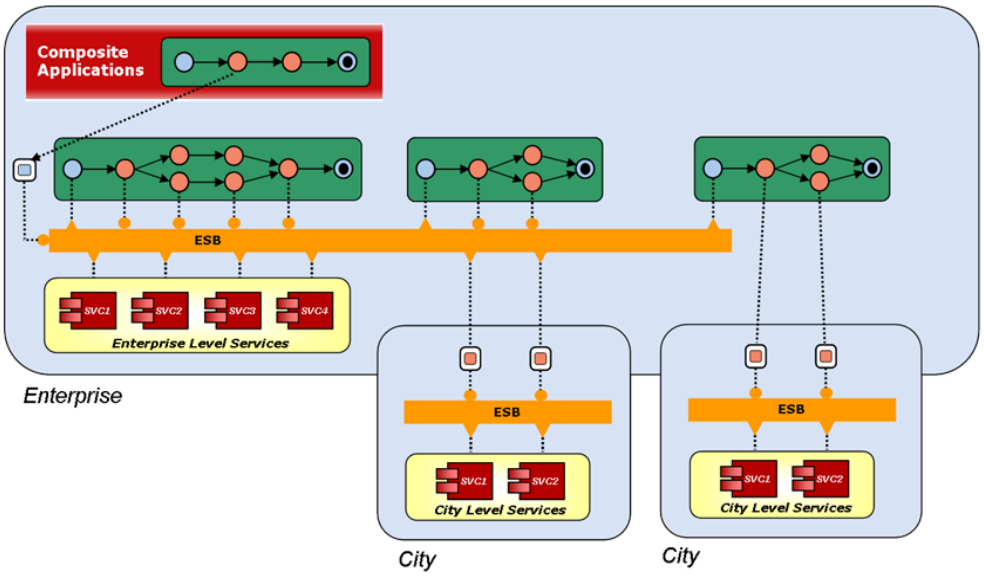
Figure 4–5 Federated Processes and Services



Federation also applies to the level of autonomy each domain has with respect to processes and Services. Processes in one domain may leverage processes that are offered as shared Business Process Services in another domain, as illustrated above. Services that represent automated business activities (tasks) within a process may be deployed in the local domains. In this scenario each domain retains autonomy in terms of how its processes are written, which tasks are fulfilled by Services, and how the Services are offered (interface, security, implementation specifications). It is most useful when a "black box" approach is desired, i.e., details of sub-process flows including current status, execution logic, etc., do not need to be known or managed at the enterprise level.

Another option for process federation is shown below in Figure 4–6. This method involves defining processes and sub-processes at the enterprise level. The Services that fulfill automated tasks may be offered in different locations, and accessed in different ways. This method is preferred when entire end-to-end process management and monitoring is needed. Each process and sub-process can be controlled at the enterprise level, while Services are managed at the domain level.

Figure 4–6 Co-deployed Process with Federated Services



There is no one correct method for process and Service deployment. A mixture of the two strategies may be used depending on process management needs.

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 a technology strategy based on a set of implementation standards and having its own architecture (**WSA**), while SOA is an Enterprise Architecture (EA) strategy. In this broader scope of SOA we find a technology agnostic definition of a Service, under which various technology strategies may be employed.

This document has shown that SOA has a broad deployment model due to the Service having substantial parts of its implementation in infrastructure. The infrastructure associated with, and largely required for SOA, is further elaborated in the *ORA SOA Infrastructure* document. SOA is also commonly employed as an integration strategy and this aspect is covered in detail in the *ORA Integration* document.

Services within the SOA definition arise in many forms, such as, encapsulated legacy application functions (non-Service enabled assets) emerging in Connectivity Services, or embedded in Services other at any architectural level, legacy enablement (Service enabled assets), composition of other 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, the 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

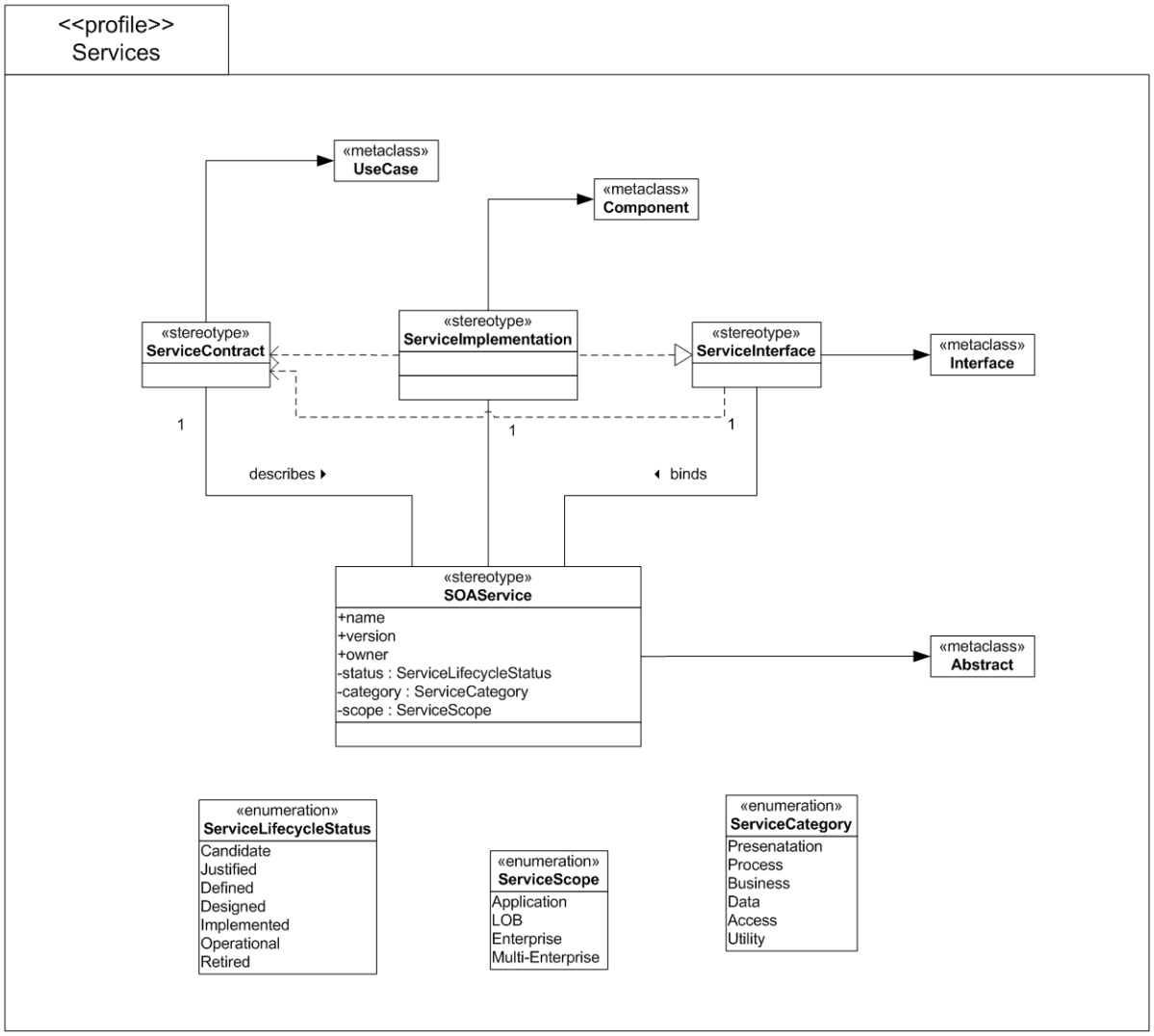
UML Profile for Services

This appendix defines and describes a UML profile for modeling Services. The UML profile incorporates the aspects of a Service that were discussed in Chapter 2 of this document.

A.1 UML Profile for Services

The diagram in Figure A–1 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 A–1 UML Profile for a Service



The diagram in Figure A–1 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, 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 Service Contract
- ServiceImplementation: extending a UML Component, realizing the operations of the Service Interface while conforming to constraints and implementing the functional requirements of the Service Contract

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.

B

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 1.5 and formalized by the service-meta model presented in Appendix A.

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 as defined in Section 1.5. 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.

B.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

B.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

B.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.

B.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.

B.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.

B.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.

B.2 SoaML

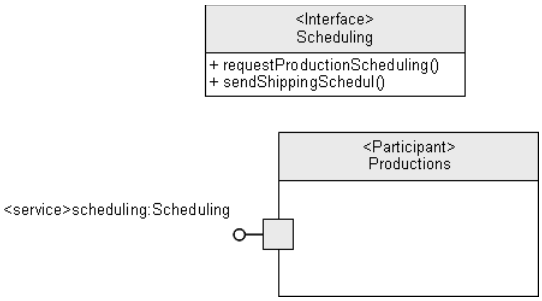
SoaML is designed to support service modeling and design providing sufficient detail to support a model-driven development approach. SoaML is based on UML 2.0 specification and provides a fully defined meta-model that uses the extension mechanisms provided by UML 2.0. The SoaML specification is available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

B.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 B–1 shows a simple example of a service.

Figure B–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.

B.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.