Oracle® Practitioner Guide

Software Engineering in an SOA Environment

Release 3.0

E14486-03

September 2010

ORACLE®

Software Engineering in an SOA Environment, Release 3.0

E14486-03

Copyright © 2009, 2010, Oracle and/or its affiliates. All rights reserved.

Primary Author: Stephen G. Bennett

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

Warranty Disclaimer

THIS DOCUMENT AND ALL INFORMATION PROVIDED HEREIN (THE "INFORMATION") IS PROVIDED ON AN "AS IS" BASIS AND FOR GENERAL INFORMATION PURPOSES ONLY. ORACLE EXPRESSLY DISCLAIMS ALL WARRANTIES OF ANY KIND, WHETHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. ORACLE MAKES NO WARRANTY THAT THE INFORMATION IS ERROR-FREE, ACCURATE OR RELIABLE. ORACLE RESERVES THE RIGHT TO MAKE CHANGES OR UPDATES AT ANY TIME WITHOUT NOTICE.

As individual requirements are dependent upon a number of factors and may vary significantly, you should perform your own tests and evaluations when making technology infrastructure decisions. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle Corporation or its affiliates. If you find any errors, please report them to us in writing.

Third Party Content, Products, and Services Disclaimer

This document may provide information on content, products, and services from third parties. Oracle is not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Limitation of Liability

IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, ARISING FROM YOUR ACCESS TO, OR USE OF, THIS DOCUMENT OR THE INFORMATION.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Contents

List of Figures

Send Us Your Comments

Software Engineering in an SOA Environment, Release 3.0

E14486-03

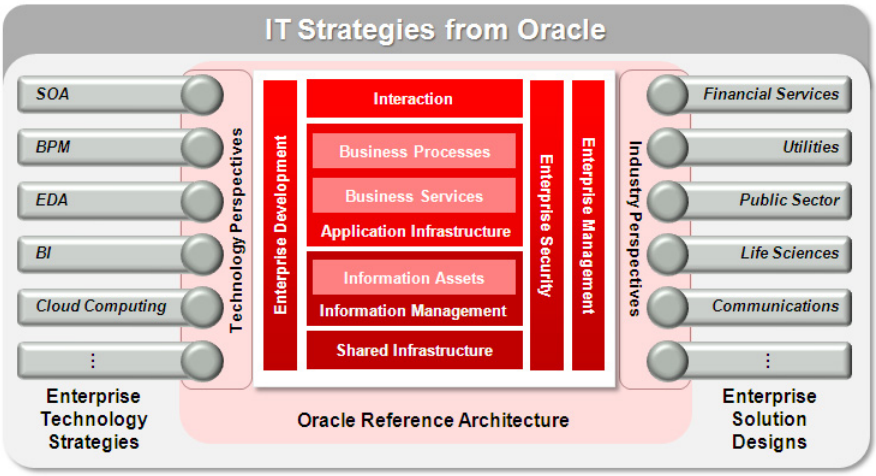
Oracle welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- 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

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

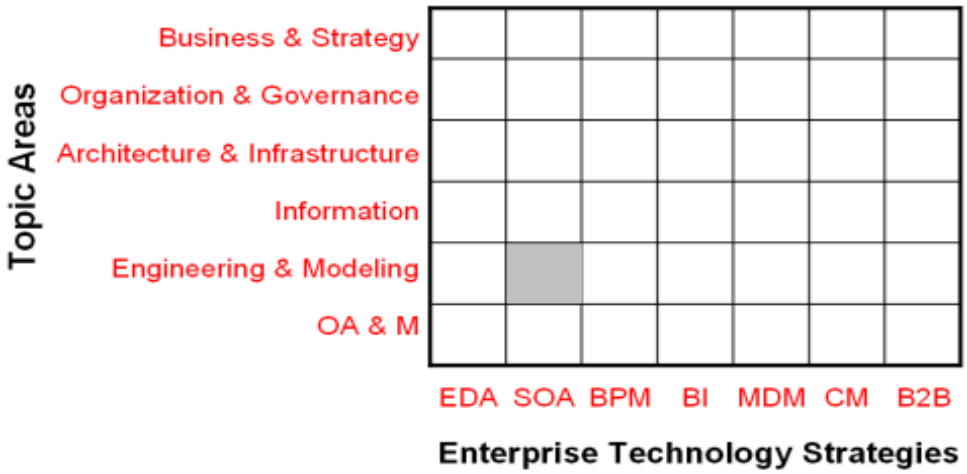
required to build enterprise-wide industry solutions. ESDs also map the relevant application and technology products against solutions to illustrate how capabilities in Oracle's complete integrated stack can best meet the business, technical, and quality of service requirements within a particular industry.

This document is 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.

Document Purpose

A Practitioner's Guide provides insight and guidance when working with a particular type of technology and address the common concerns faced by enterprises and practitioners.



This Practitioner's Guide provides an approach for delivering projects within an SOA environment. It identifies the unique software engineering challenges faced by enterprises adopting SOA and provides a framework to remove the hurdles and improve the efficiency of the SOA initiative.

Audience

This guide is intended for project managers, enterprise architects, application architects, developers and other stakeholders in delivering projects in an SOA environment.

Document Structure

This document is organized into the following sections.

Chapter 1 - provides an overview of the Oracle Service Engineering Framework which covers an approach to Service engineering.

Chapter 2 - provides a number of key concepts that form the basis for the rest of the guide.

Chapter 3 - provides a high-level approach to Service analysis covering SOA requirements management, Service Identification, and Service Release Planning.

Chapter 4 - provides a high-level approach to Service Delivery covering Service Definition, Service design, and Service implementation.

Chapter 5 - provides a high-level approach to Service management.

Chapter 6 - provides a concise summary of the document.

Appendix A - provides a quick reference on where to find further information.

How to Use This Document

This document should be read by everyone that is interested in learning about SOA Service engineering. The first two chapters provide an overview and key concepts, while the chapters that follow provide more detail on the three phases of Service delivery.

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 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 document series, the word appears with its initial letter capitalized ("Service"), while all other uses of the word appear in all lower-case (e.g. "telephone service"); exceptions to this rule arise only when the word "service" is part of a name, such as, "Java Message Service" ("JMS"), "Web Service", etc.

1 Introduction

Many of the early SOA projects were focused around Service enablement of business functionality in existing systems. As corporate SOA maturity levels increase, so does the need for developing and composing Service oriented applications to assist with closing the IT gap. Many enterprises are still attempting to use traditional delivery methodologies for SOA. However, these methods are designed to deliver projects that do not consider the requirements of the business outside of the scope of the project in question. These methods are too narrowly focused and need to be adjusted to enable SOA.

Further, the extensive scope that a SOA may have across an enterprise requires that a repeatable process and sound engineering disciplines be applied through delivery cycles. This is especially true if external or offshore resources are to be utilized. Many enterprises struggle with inconsistency across deliveries for projects that need to coexist. This problem is also an inhibitor to reuse and agility and can be addressed by the adoption of a sound enterprise class service engineering & modeling framework.

Today, many customers adopting SOA have been addressing some of the issues of delivering SOA projects independently. For instance, there are several different approaches to performing Service Identification. There are also different approaches to defining the Service Contract. However, independent solutions to specific problems do little to help the enterprise go from start to finish through the Service engineering process. Part of the problem is that a framework or methodology is required that connects these processes. When you examine any of the traditional software engineering methodologies, you will notice a connectedness between the activities, and a flow of information required to make the method effective. The same connectedness, and information flow is required for a Service engineering framework

The value comes from beginning with new projects or applications that will add business value. These projects need to follow a process that begins with their own requirements and business processes, where reuse benefits are identified from the existing set of requirements and business activities, and then examined against existing assets in the enterprise. Service candidates are then justified, and then engineered into Services from either existing assets, or by engineering new Services. Alternatively, requirements are fulfilled through the discovery of existing Services that have resulted through the engineering of past projects following the same process. The act of harvesting Services by examining the existing enterprise assets and processes for the sake of building a Service catalog is simply counterproductive.

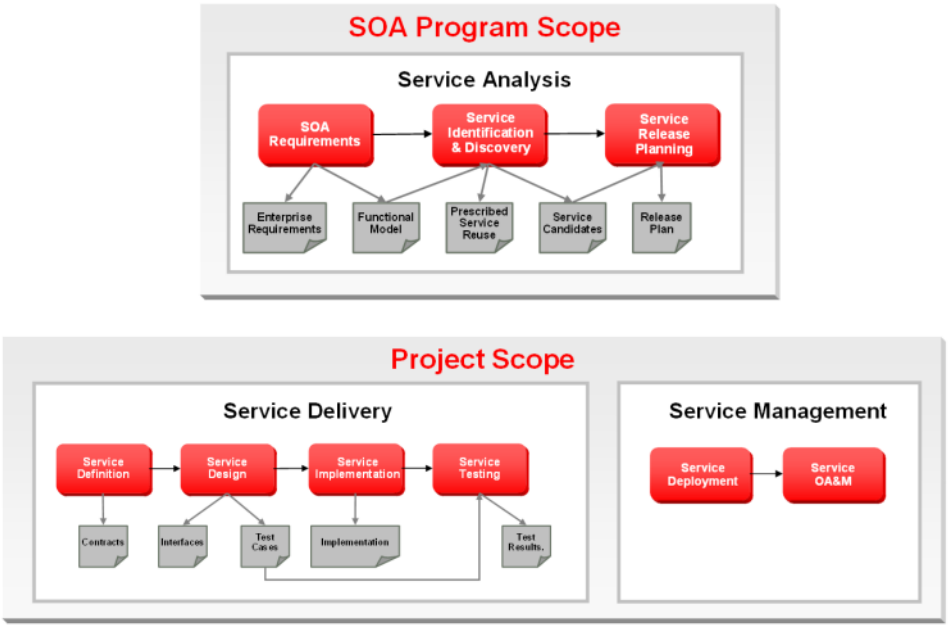
1.1 What is the Oracle Service Engineering Framework?

The Oracle Service Engineering Framework is an engineering approach for delivering projects within an SOA environment. It identifies the unique software engineering challenges faced by enterprises adopting SOA and provides a framework to remove

the hurdles and improve the efficiency of the SOA initiative. It identifies and resolves the unique software engineering challenges encountered by enterprises adopting SOA. It complements traditional delivery methodologies by defining the engineering disciplines required for effective and consistent service delivery.

The Service Engineering Framework addresses activities at both the program and project scope to consider the requirements of the business outside of the scope of a single project.

Figure 1–1 Oracle Service Engineering Framework



Topics covered at the program scope include:

- **SOA Requirements Management** - Provides a process for harvesting requirements in a manner that naturally facilitates service identification and discovery.
- **Service Identification & Discovery** - Establishes the procedures around identifying Service candidates, as well as discovering reuse candidates from the existing Service catalog. Takes the process from identification and discovery, through the justification processes required to determine if an existing Service can be viable for reuse in the proposed manner, or if the proposed Service Candidate should be realized as a shared Service.
- **Service Release Planning** - Provides the groundwork necessary for planning for project and Service deliveries within an SOA.

Topics covered at the project scope include:

- **Service Definition** - Takes the identified Service Candidates and defines the Service boundaries and resulting Service Contracts.
- **Service Design** - Provides the best practices and procedures required to design Services from the Service Contract and produce the Service interface.

- **Service Implementation** - Provides the guidelines for effectively and efficiently developing shared Services.
- **Service Testing** - Defines the strategy that should be taken with respect to ensuring the appropriate level of quality for delivered shared Services. The information obtained through Service testing is also used to enable Service Deployment.
- **Service Deployment** - Defines the guidelines and practices that need to be considered when deploying Services into a shared environment.
- **Service OA&M** - Covers the operation, administration and maintenance guidelines required for supporting the SOA's operational environment. OA&M is not simply about keeping the environment operational, but enabling reuse and evolution, in addition to measuring the SOA's adoption and success.

2

Key Concepts

2.1 Service Definition

A Service can be described as a way of packaging reusable software building blocks to provide functionality to users and to other Services. A Service is an independent, self-sufficient, functional unit of work that is discoverable, manageable, and measurable, has the ability to be versioned, and offers functionality that is required by a set of users or consumers.

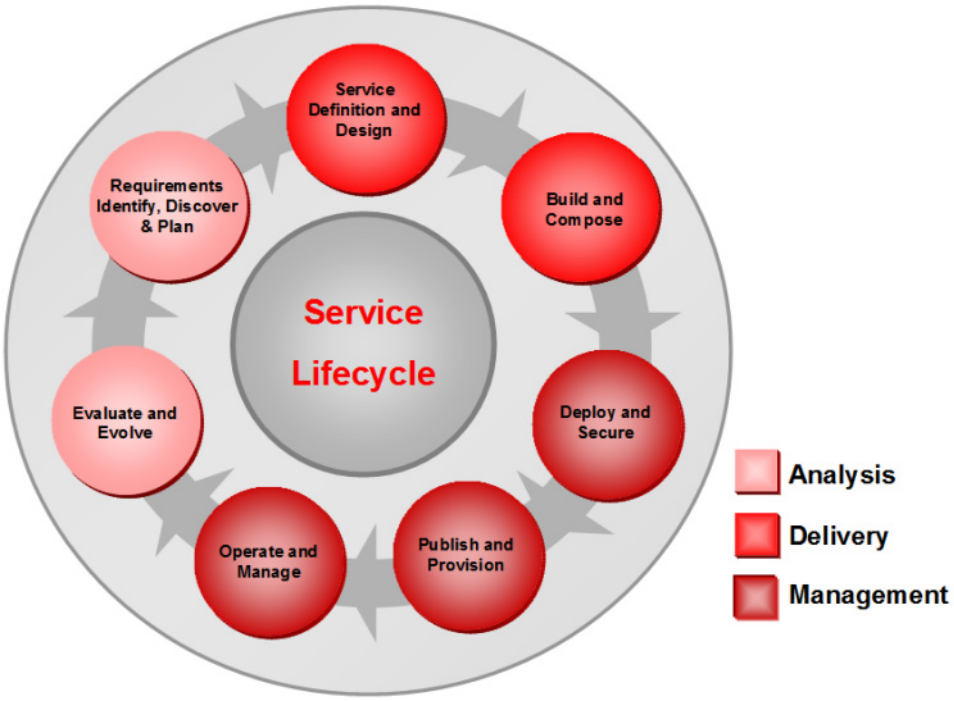
A logical definition of a Service consists of three components: A Contract, its Interface and the Implementation.

See *ORA SOA Foundation* document for a more detailed description of a Service and its scope.

2.2 Service Lifecycle

The Service lifecycle tracks the Service from inception through retirement, and includes the Service's evolution through multiple Service versions. The Service lifecycle consists of two views. The first view is the set of phases that a Service goes through, and the second is a more detailed view consisting of the states that the Service transitions through. The Service lifecycle phases are depicted below:

Figure 2–1 Service Lifecycle Phases



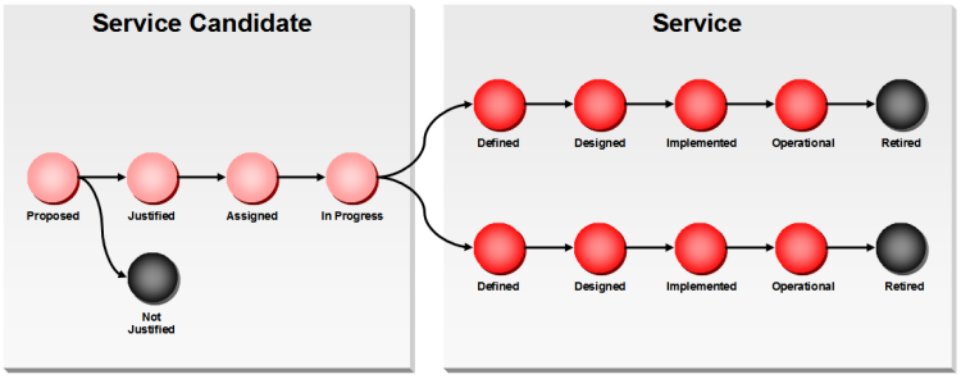
A Service begins its "life" as a Service Candidate which is identified in the analysis phase. Analysis for most projects begins with the requirements gathering process, and proceeds through Service identification and discovery. Analysis completes with release planning. In the event that an existing service will be extended or "Evolved", the Analysis phase begins at the tail end of the lifecycle.

Service delivery takes a Service in candidate format and delivers the contract, interface, and implementation, or new versions of the same. Service delivery concludes with the Service artifacts in a state deemed ready for deployment into the operational environment.

The Management phase begins with the Service or new Service version being deployed into the operational environment, where it is then published and made available to consumers. It is then operated and managed until a new revision is necessary or the Service is to be retired.

The second view of the service lifecycle depicts the various states the Service travels through. This view is depicted below:

Figure 2–2 Service Lifecycle States



A Service begins as a Service candidate, which is initially proposed when it is identified. It is then either justified for realization or in the event of a failed justification the requirements linked to the Service Candidate become the responsibility of the project rather than a shared Service Delivery. Justified Service Candidates are then assigned to a delivery team for realization and remain "In Progress" until delivery commences.

A Service Candidate may result in the definition of more than one Service. Service boundaries are determined during the Service Definition phase which is where the Service Candidate transitions from Service Candidate to Service. Once Service Definition has completed the Service is in the "Defined" state. It then goes through Service design and implementation where it passes through corresponding states. Once the Service has been deployed into the operational environment it enters the "Operational" state. A Service may eventually become retired when it is no longer relevant within the SOA. In the event that revisions to the Service are necessary, depending on the magnitude of the change, the Service may go through the Service candidate lifecycle again in the definition and justification of major revisions to the Service.

3

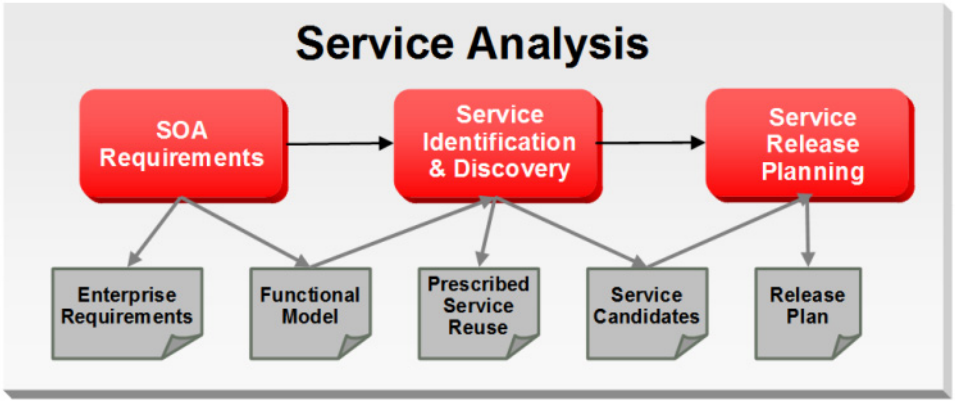
Service Analysis

As with traditional software engineering, Service engineering also begins with requirements. In fact, there are a number of parallels that can be drawn between Service engineering and traditional software engineering practices. The main difference however is thatSservice engineering evolves and creates interconnectedness between projects participating in an SOA. Traditional software engineering projects typically begin and end within their own lifecycle and have very little dependencies outside of the project itself.

It is important to highlight that SOA adoption is an investment that provides increasing returns over several projects. The first project participating within an SOA will more than likely suffer from some additional overhead and loss of productivity, as education and practice is required to refine the procedures around SOA and Service engineering, more importantly though, each project is developed within a larger picture. This means, that additional engineering and discipline is required to effectively deliver projects within an SOA, which results in an increase in early project delivery time. Over time, this investment can be significantly overcome with the advantages gained by a mature SOA, such as business agility achieved through application and Service composition.

The Service Analysis phase of the Oracle Service Engineering Framework consists of three main sets of engineering practices: SOA Requirements Management, Service Identification & Discovery, and Service Release Planning.

Figure 3–1 Service Analysis Phase



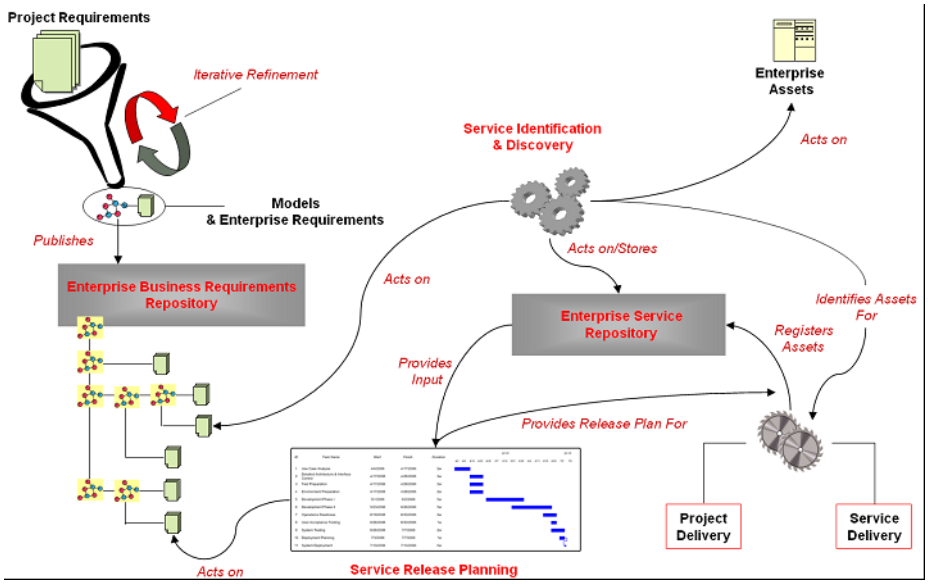
- **SOA Requirements Management** - consists of gathering captured project requirements and then refining and classifying these requirements at the

enterprise level. The information gathered within the process provides a key enabler for the rest of the analysis processes. The value of the information gathered through SOA Requirements Management grows over time.

- **Service Identification & Discovery** - provides an analytical method for identifying Service candidates together with discovering existing Services and Service candidates. The result is a set of Service Candidates that have been justified and will be scheduled for delivery through Service Release Planning, as well as Services and Service candidates that have been prescribed for reuse by the project.
- **Service Release Planning** - is an iterative process that schedules SOA projects and Service Candidates for delivery. The dependencies between projects and Service candidate delivery cycles are also managed through Service release planning through to deployment. Service release planning takes care of resource constraints through priority-based contingency planning.

The relationships and inforomation flows between these set of practices can be seen in Figure 3–2

Figure 3–2 Service Analysis Flows



3.1 SOA Requirements Management

SOA Requirements Management represents an extension to, rather than a replacement for, traditional requirements gathering techniques. However, more interaction is required between business and IT. In order for the process to be successful, the business will no longer be able to simply draft requirements and "throw them over the wall", as an iterative procedure is required.

A few of the key differentiators of SOA based requirements management are:

- Project requirements are refined into enterprise assets.
- Project requirements and modeling techniques are used to construct enterprise level functional models

- Requirements are classified against an enterprise-level functional model rather than at the project level
- Enterprise requirements have an independent lifecycle
- Enterprise requirements are managed centrally

3.1.1 Traditional Approach to Requirements

The traditional approach to requirements gathering is fundamentally focused on the requirements for a single application (or project). Areas where integration is required with other applications are covered off as requirements indicating how the new application will need to interoperate with the existing. This method works very effectively for constructing applications but has always resulted in problems with enterprise planning, integration and enterprise scaled reuse.

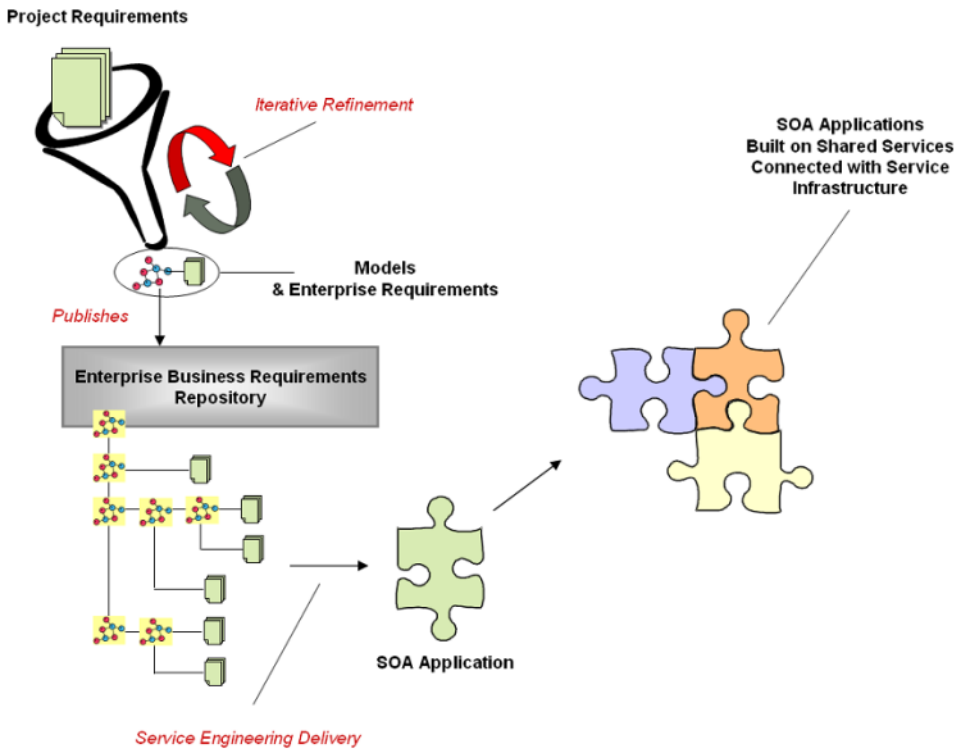
This approach breaks down when applications need to be developed as part of a larger SOA, as these applications need to be planned at the enterprise level, rather than in a self-contained silo. In the traditional approach, the business would construct their objectives and goals for the project and work with system analysts (and sometimes architects) to develop detailed requirements and/or Use Cases. These requirements are then given to IT to deliver in the form of an application. The business's involvement was typically minimal, where they sometimes had very little interaction with the project until the onset of user acceptance testing. There are obvious variations to the approach used between enterprises, but the result was typically the same: An application that was built within a silo, with little consideration to existing or future use within the enterprise. The application is then stood up in production and managed independently.

From the enterprise perspective, some semblance of integration and reuse was achieved by establishing a n-tiered application architecture and prescribing this architecture to the delivery teams for individual projects. This technique simply drove the enterprise level integration between applications and data into the applications themselves, rather than addressing it at the enterprise level. The reuse achieved was typically library based, and runtime reuse through integration, resulted in applications being chained together through tight-coupling.

3.1.2 A New Approach is Needed for SOA

In order to achieve effective SOA, applications simply cannot be developed independently of one another. Further, SOA applications utilize shared Services which are not owned by any single application, consist of their own lifecycle and are managed independently. In order to be effective in managing requirements within an SOA, applications have to be aware of the requirements of existing applications, in-flight applications, as well as proposed applications. This technique is one of the fundamental enablers for Service identification and discovery, as well as the centralized information required to achieve service release planning. The traditional method of requirements management is not designed to handle these demands.

Figure 3–3 SOA Requirements



Once the functional models have been defined, and classified, the requirements are refined and classified around the functional models. The result is a taxonomy of requirements based on functional models, rather than requirements grouped by specific areas of application function. This allows duplicate or overlapping requirements to be identified, in addition to providing a basis for looking at requirements with a potential wider audience within the enterprise. This approach therefore provides a natural avenue for enabling the Service Identification and Discovery process.

The classification of functional models and requirements is achieved iteratively with the business, where the business provides input regarding the overall business process, and the delivery team and SOA Leadership team develop the models and classifications which are then cross-checked with the business.

Finally, as more and more applications are delivered using this model, the value of the information available for SOA deliveries will grow exponentially. A clear management of the taxonomy is important in retaining the value of the information stored, as it will grow rapidly with each project delivery.

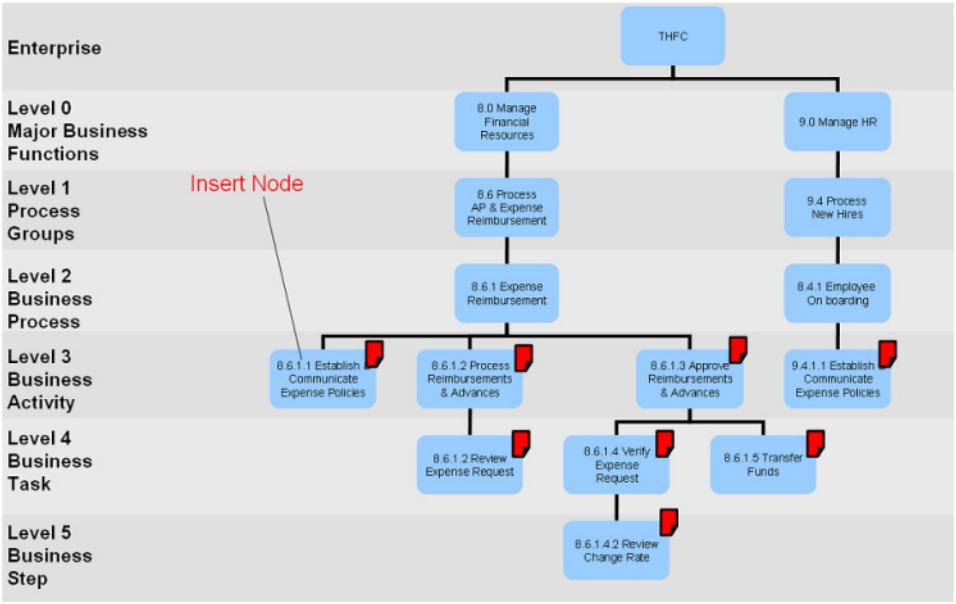
3.1.3 Requirements as Enterprise Assets

Scoping requirements at the enterprise level, rather than at the application or project level enables project architecture and planning exercises to remain connected to the overall enterprise SOA. This connection also provides a natural enabler for Service discovery, identification, and Service Release Planning.

Managing requirements centrally, at the enterprise level, also demands increased discipline in order to prevent the data collected from becoming disorganized where it

will lose its effectiveness. This means that an effective and consistent classification system is required to ensure that functional models and their respective requirements are always classified correctly. Classification around functional models is an effective way to accomplish this. In this approach top level functional models create a branch of an enterprise requirements repository.

Figure 3–4 Sample Requirements Classification



It is a good practice to create a high level taxonomy structure up front when establishing this process rather than building it out as applications are constructed through the process. There is little doubt that the taxonomy will grow over time, but at least the general framework and structure has been established up front which sets the process for the first application.

Another reason for tracking enterprise requirements is that they can be linked to the applications and Services that implement them. In the case of Services, this is especially useful as requirements form the basis of the Service Contract definition in a rough form.

The functional models are used to group requirements within the Enterprise Requirements Repository. These models also represent the branches within the repository. In a well structured requirements repository, the complete picture of linked enterprise functional models would be represented. In reality there will be holes in the picture unless the organization is willing to do a large amount of work to model the entire business (unrealistic), or the organization is starting from a completely blank pallet (not very likely).

The reason for using functional models to classify requirements is for purposes of navigation and scale. Navigating through the functional model should be quite natural where you can easily find an end point by navigating through a tree-like structure (think binary search, vs. linear search). Once you get to the branch containing the relevant functional model, the requirements for the model are attached to either the entire model itself, or specific activities within the model. This method also makes it

easier to identify duplicate requirements, and is scalable to allow for large numbers of requirements.

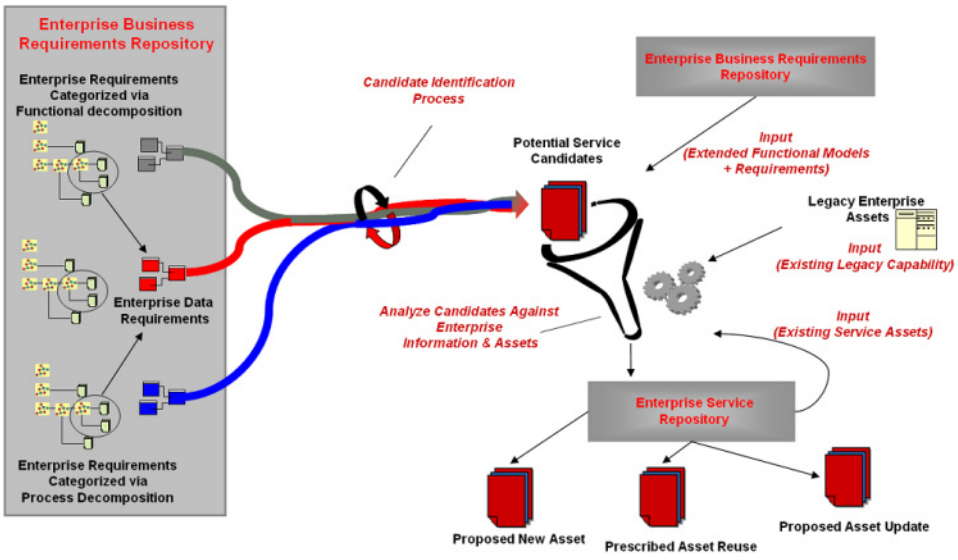
A recommended approach to beginning the requirements gathering process is to develop the functional models, process models and/or data models for the business with respect to the proposed application or project.

- Functional models assist in defining the business functional groupings in a hierarchical structure. These models also identify data requirements since each functional area has specific data needs.
- Business process models capture business processes which also have specific data requirements
- Data models identify semantic communities, data entities and their relationships, and data usage patterns.

These models are then iterated over to ensure that the resulting set of models represent the enterprise as well as the needs of the project or application. Once the models have been established, develop and categorize the requirements around these models. Then refine and reclassify the requirements to the business in a similar fashion. This work should be minimal as the classification of the functional models prior to developing the requirements will set you up for the correct requirement classification. Iterate over these steps with the business until it is agreed that the correct representation has been established.

3.2 Service Identification & Discovery

Figure 3–5 Service Identification Flows



Without Service Identification & Discovery, SOA is not possible. Service Identification deals with the procedures and guidelines that an enterprise adopts to identify new shared service candidates. Discovery is as equally important since it provides the mechanism for which existing shared Services can be efficiently discovered when the need arises within a new project or application. These two topic areas have been referred to as the "secret sauce" of SOA, and share an obviously natural relationship to

one another. In the process of analyzing requirements, new Service candidates need to be considered on two angles. Either they have already been realized in the form of a shared Service; or they are a potential candidate for a shared Service.

- **Service Identification** - begins with analysis of requirements and ends with the identified set of Service Candidates. Service Candidates, represent the set of business functions that have been recommended for development as a Service. A Service Candidate does not necessarily map one-to-one to a Service, as Service candidates could be realized through the creation of one or more Services. This determination, however, is outside the scope of Service Identification, and is part of the Service Definition procedures. The Service Identification and discovery process identifies Service Candidates that have been identified as being viable shared Service Candidates that need to be realized, rather than just proposed Service Candidates.
- **Service Discovery** - is the process where analysis is done against requirements, business processes and existing enterprise assets, to determine where Services already exist that may be viably reused to fulfill some of the requirements that need to be addressed by a project or application. Service discovery is a natural extension of the Service Identification process, where by working your way through proposing Service candidates, existing Services will be discovered.

Service Identification & Discovery is a fundamental enabler of SOA. Therefore, a repeatable and pragmatic approach is required for the successful adoption of SOA. There are a number of different approaches to performing Service Identification & Discovery, all of which have relied upon a great deal of "gut" feeling. The key enabler to these processes however, is the availability of information, and an understanding of where to look to get the information. Human instinct is always going to be a fundamental part of any analysis exercise, when not all of the required information is available to make a decision. However, the more information we can make available to the process the more repeatable, consistent, and effective it will be.

In order to achieve effective Service Identification & Discovery (SID&D), these processes cannot be simply worked through in a standalone manner. If this is attempted, there will simply not be enough information available to ensure somewhat consistent results. The processes and guidelines around SID&D are a part of the larger Service Engineering Framework, which provides a connected methodology and best practice for achieving effective delivery of SOA based projects. The framework covers a number of topics core to delivering SOA projects where information is collected and shared throughout the framework. This puts a pretty strong requirement for an efficient information management solution, but the investment in getting the process right and having the right tools for the job will pay large dividends over the long haul.

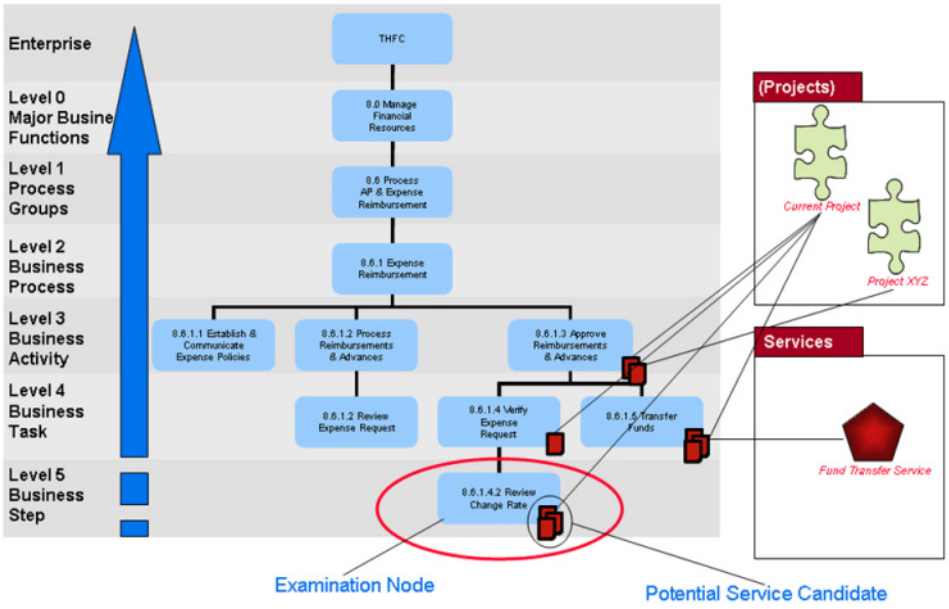
3.2.1 An Analytical Approach to Service Identification & Discovery

The process begins with identifying potential Service Candidates as well as discovering reuse candidates. Potential Service Candidates have not yet been justified for realization. Utilizing modeling techniques there are a number of approaches to identifying potential Service Candidates, and reuse candidates.

- **Functional Driven** - A functional model can be used in one of two ways. A typical functional model for Service Identification stems from business domain analysis and building business function models, which can then be utilized to identify which applications support the relevant business functions. These models can be de-composed and aligned with individual project requirements to highlight candidate Services. Another typical functional model approach is turning project

requirements into a functional model and utilizing a Service engineering scoring tool to highlight candidate Services. Both approaches can be utilized in identifying which applications can provide the functionality required or the gaps where functionality needs to be built.

Figure 3–6 Example - Functional Model After Analysis



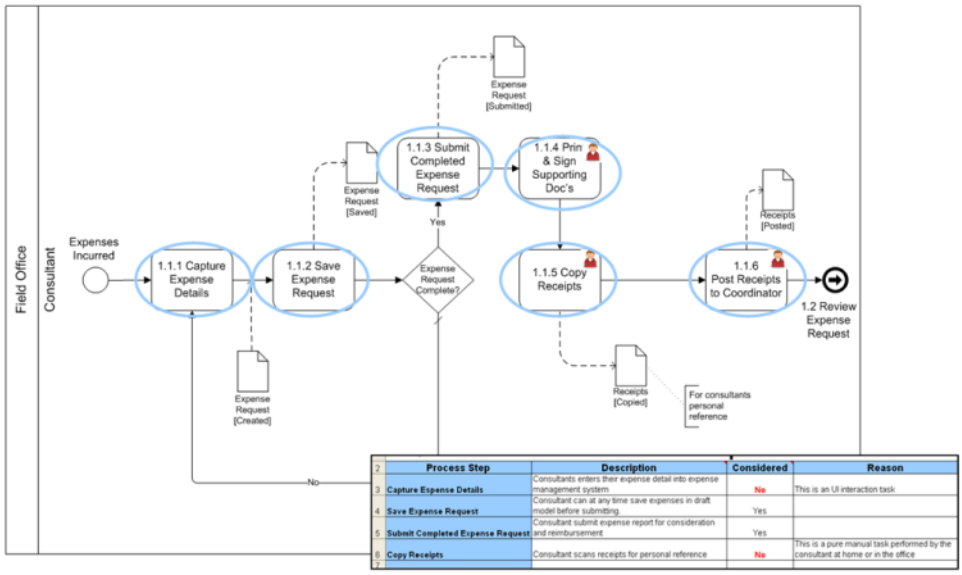
When the SOA Requirements process is complete, all projects stemming from the project are still linked to the project (whether duplication exists or not).

The best approach is to begin with the bottom most nodes in the functional model and work your way upward. The reason for this approach is that you want to maximize the chance of discovering the finest grained Services first. That way Service composition analysis, which is performed within Service Definition, is a much more natural exercise. Further, the functional model is a viable input into the Service Definition process to support the boundary analysis process, especially when a fairly course grained candidate has been approved for realization.

The rest of the process continues by iterating over each node with requirements attached that have been linked to the project and analyzing the model for the purpose of Service Candidate Identification and Service Discovery.

- Business Process Driven – A business process driven approach for Service Identification stems from utilizing a business process model which is decomposed into candidate business Services operations. Then the appropriate information payloads are defined using various data models. (See Example 3–7)

Figure 3–7 Decompose Business Process



Both of these approaches have information requirements where potential data Service candidates can also be identified as well as reuse data Service Candidates. A typical data model approach for Service Identification stems from utilizing models such as entity relationship models, semantic communities, and data glossaries to highlight candidate Services.

Candidate Services are checked against the enterprise repository for duplicates. If duplicates are found then a check is done to see if the duplicate is an exact match, or a partial match. If a partial match is found, or no match is found, then the candidate Service is registered in the enterprise repository. The candidate Service is then examined for applicability beyond the project's scope. If a candidate Service is determined to be relevant beyond a project's scope then it becomes a potential Service candidate, if no match was found; a potential reuse candidate if an exact match was found; or a potential asset extension candidate if a partial match was found.

Once reuse and potential Service candidates have been identified they are funneled through a reuse validation or Service Candidate justification processes (respectively). A reuse validation process is used to determine whether a proposed reuse candidate will be authorized for reuse by the application. If the candidate is authorized then it is prescribed for reuse with the project, otherwise it is rejected and the candidate can be proposed as a new potential Service Candidate, or handled within the project's delivery.

A Service Candidate justification process analyzes a potential Service Candidate and determines if it should be realized as a shared Service or not. A scoring system is used to assist with the analysis. The weights or scores as well as the scoring criteria are customizable to fit the SOA adopting enterprise's preferences and priorities. Data from existing enterprise assets as well as legacy systems are used in the analysis. If a Service candidates or Service extension candidates are justified then they are scheduled for realization.

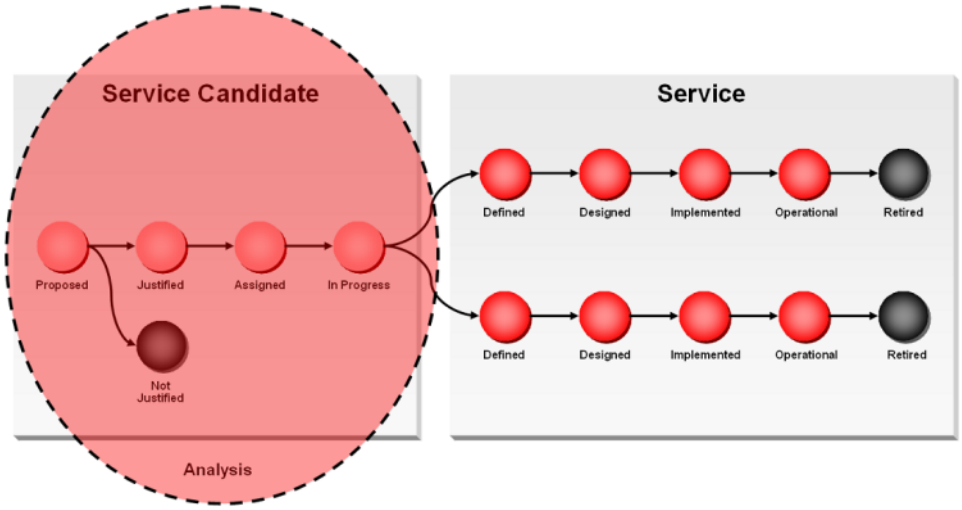
3.3 Performing Service Identification & Discovery

Effective Service Identification & Discovery is achieved by having the correct information available to analytically make informed decisions, rather than relying on instinct or gut feel.

Recording Service Candidates, regardless of whether they are realized as shared Services or not, is an important enabler for SID&D. Service Candidates that have been realized provide the linkage from requirements to the realized Services and back, which is useful in doing reuse validation analysis. Further, unrealized Service candidates that continue to be proposed as potential Service candidates by additional applications provide an indication that it may be time to realize the candidate as a Service.

An enterprise develops a balanced scoring system that meets their needs and priorities. The system should be refined over time as the enterprise matures with SOA adoption. In order to do this, the scoring system should be reviewed periodically (eg: once a year), to ensure it is still capturing the needs and priorities of the enterprise with respect to SOA.

Figure 3–8 Service Candidates States



Service Candidates progress through a series of states throughout their lifecycle. When a Service Candidate is created it begins in the "Proposed" state. The candidate then is pushed through the justification process. If justification is received the candidate is set to the "Justified" state and is ready to go through the release planning process. If justification is denied, the candidate is set to "Not Justified" where it remains until it is encountered by another project and sent through the justification process again. Each time a candidate is passed through the justification process a counter is incremented. This counter should add justification influence for candidates that have failed to be justified multiple times. Once the release plan is completed for the project where the Service Candidate will be realized, the candidate enters the "Assigned" state as the release planning effort will assign the delivery teams. The candidate moves to the "In Progress" state when the delivery team commences work on the asset. The candidate then becomes a Service once it passes through the Service Definition process. Service Definition may result in breaking the candidate down into one or more Services.

A candidate may be versioned when in the In Progress state. New versions have their own lifecycle and begin in the "Proposed" state and go through their own independent lifecycle.

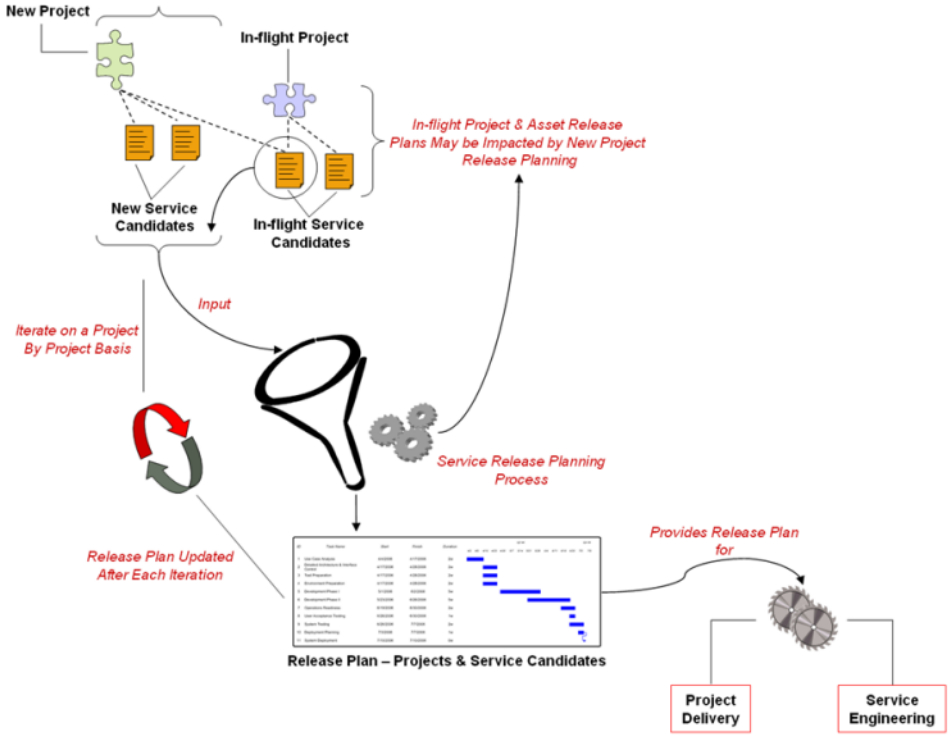
For a more detailed description of perform Service Identfication see

3.4 Service Release Planning

As with any organization, release planning is an exercise in enterprise scalability. This means that as more and more in-flight SOA projects are progressing in parallel within an SOA environment, an increasing demand is placed on the need for effective SOA Release Planning. In order to be effective, Service Release Planning must be efficient, scalable, and fair.

Efficiency and scale are achieved through a combination of having the correct information available to make informed decisions, as well as having this information organized in a way that it is readily available when decision making analysis requires it. The enterprise repository plays a key role in bringing the information together efficiently through an organized and linked taxonomy.

Figure 3–9 Service Release Planning Flows



Service Release Planning governs the delivery of both SOA projects as well as shared assets such as Services and shared business processes. The release plan is a living document that contains the high-level release information for all in-flight projects and shared assets. As entities on the schedule are delivered into operational state, they are removed from the plan. When new projects and assets are introduced they are added to the plan. The release plan is constantly changing to represent the current state of in-flight SOA deliverables.

As with all delivery efforts, resources are required to deliver projects and shared assets, which imposes a capacity constraint. One of the primary purposes of Service Release Planning is to work through capacity constraints for projects requiring the same shared, but yet to be delivered, assets with conflicting schedules and resource requirements. In these situations the Service Release Planning team utilizes the information available as well as a predetermined process in order to prioritize the release of shared assets and SOA projects. This team needs to have the authority to prioritize the delivery of certain projects and assets over others, and any schedule that is still on the release plan may be impacted by the introduction of higher priority projects or assets.

Service Release Planning continues in an iterative manner as long as there are in-flight SOA delivery activities underway. The release plan continues to evolve with the addition of each new SOA project introduced. Each iteration begins with the introduction of a new project. Before being considered for Service Release Planning, the project should be completely through the SOA Requirements Management process as well as the Service Identification & Discovery procedures. At the point where release planning begins the project will be associated with the set of new Service candidates, and in-flight Service Candidates, as well as the Services that have been assigned for reuse by the project. If there are no new Service Candidates to be realized or in-flight Service Candidates that will be utilized by the project, then the release plan for the project is simply the project specific release plan as there are no dependencies with new or in-flight Service Candidates.

When new Service Candidates are identified and attached to a project, or existing in-flight Service Candidates are connected to the project, then the project and the connected assets must go through an iteration of the Service Release Planning process. The process analyzes the available resources and allocates them to a schedule. If resource constraints are identified the process determines the course of action to be taken to ensure the needs of the enterprise are met. Contingencies may include delivering the shared asset within the project delivery using the project's delivery resource; partially delivering an asset to satisfy the immediate project needs; or rescheduling lower priority assets or projects to facilitate the needs of higher priority projects and assets.

After each iteration, the release plan is updated to reflect the current plans. Delivery teams for shared assets and projects then proceed according to the enterprise service release plan. Operations must also be kept in synch with the release plan so that they may properly prepare for additional projects and Services that will need to be hosted in the environment as delivery activities complete. The release plan itself, is not a detailed project plan, but rather a high-level release schedule which tracks the dependencies between projects and assets as well as high-level delivery milestones (eg. phases). The release plan if adhered to by the various delivery teams involved ensures that the dependencies established between deliveries are received by the time they are needed.

3.4.1 Performing Service Release Planning

Effective Service Release Planning is achieved by having the correct information available to analytically make informed decisions, rather than relying on instinct, gut feel, or the deliberation skills of the various project teams (or Lines of Business).

The project is introduced as an enterprise asset during the SOA Requirements Management process. During that stage, enterprise requirements and functional models are established in the enterprise repository and associated with the project. The project then goes through the Service Identification and Discovery procedures where Service Candidates (and Services), which will be utilized by the project, are identified

and associated with the project. As the project moves through the Service Release Planning process it receives an enterprise priority score which rates its importance with respect to the enterprise, as well as its high-level scheduling information. Service candidates that are associated with the project are assigned delivery dates by the project delivery team, which are then used in the release planning process to determine if the requested dates for shared assets can be met. If not, the contingency process determines the course of action. Once through the Service Release Planning process the project will be assigned high-level schedule information. The schedule for a project or shared asset can be impacted by future release planning exercises for projects that have higher priority. A contingency plan may cause the release of some shared assets to be delayed and impact the projects that depend on the release of these assets.

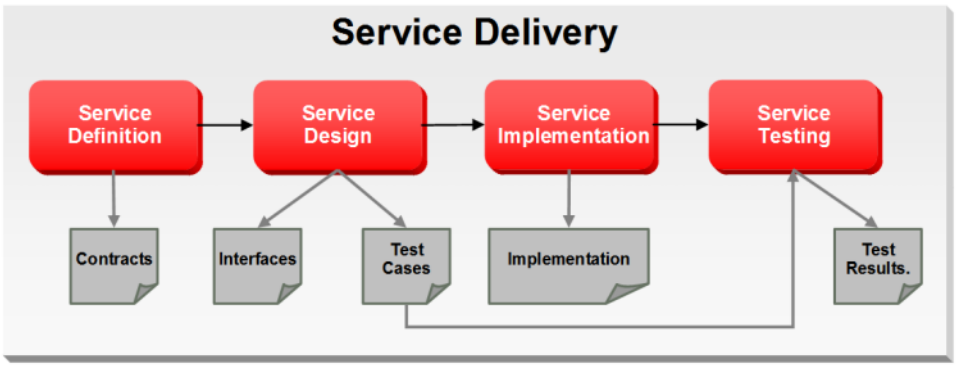
Projects remain associated with the shared assets that they reuse after they become operational. This provides additional value as these associations can be used as a measurement for reuse. Further, the release schedule information attached to the project can be used to measure the cost savings achieved through reuse. Tracking projects through their lifecycle by utilizing an enterprise repository ensures that all of this information is easily accessible and measurable. It also provides a valuable operational tool as all dependencies between shared assets can be tracked.

4

Service Delivery

Service Delivery consists of the core delivery engineering activities taking a Service candidate and molding it into one or more Services. Service candidates that enter Service Delivery have been justified for realization and scheduled for release.

Figure 4–1 Service Delivery Phase



- **Service Delivery** begins with **Service Definition** which primarily determines Service boundaries as well as the construction of the Service contract. Service Contracts include the linked requirements coming from the Service Candidate, but also may be influenced by enterprise and LOB scoped contracts that are appended to every Service based on its intended scope.
- **Service Design** then acts upon the Service contracts to develop the Services' interfaces. The process of defining a Service interface is much more involved than simply coming up with the input and output for the Service. Service design analyzes the contract from the consumer's perspective, and is influenced by factors such as scope (enterprise, LOB, application, etc), message exchange patterns (MEPs) as well as non-functional requirements such as expected volume, and response time requirements (specified in the contract).
- **Service Implementation** ensures that all aspects of the Service contracts are implemented and upheld through the delivery of business logic as well as the deployment to Service Infrastructure. The implementation must faithfully realize the Service Contract and interface which are defined through Service definition and design. Both functional and non-functional aspects (such as latency, throughput, availability, etc) of the contract must be fully realized. If some aspects of the contract cannot be realized for any reason, then the contract must be updated to reflect the true reality of the Service.

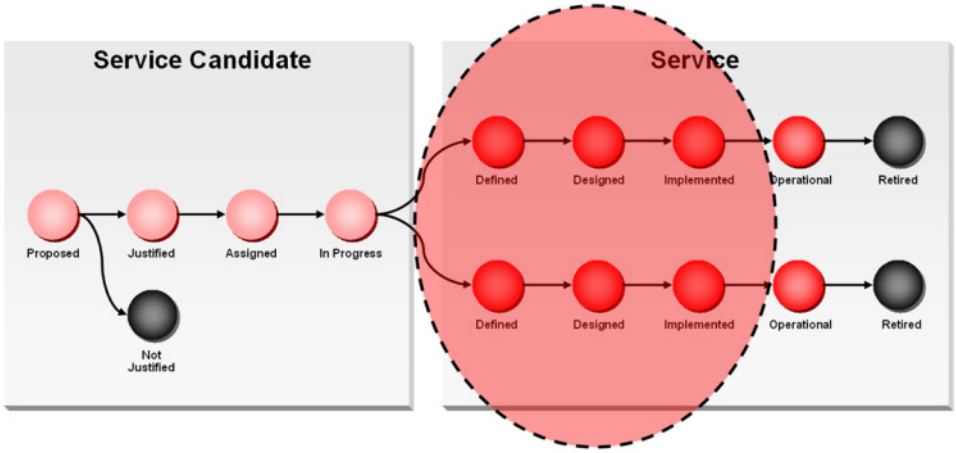
- **Service Testing** plays a role throughout the entire Service delivery cycle. After Service definition, the test criteria can be established. Once Service design has completed, the test cases can be written against the Service contracts. Finally, during Service implementation the test cases are executed to ensure expected quality, and adherence to contract is achieved prior to releasing the Service into production. Service testing must be based off the contract, rather than the traditional approach of dealing with implementation details. Since all Services must have a clearly defined contract, the contract provides the basis for the Service's test criteria.

Tools also play a significant role throughout Service Delivery, as well as throughout the entire Oracle Service Engineering Framework. In particular to Service delivery, a development environment which can integrate to Service infrastructure such as the Service registry and the enterprise repository has significant advantages. Composition tools may also be used to deliver Services through composition which may decrease delivery time, and increase reuse.

4.1 Service Definition

Service Definition covers the set of activities required to effectively define Service contracts from a Service candidate. The Service contract provides the detailed description of both functional and non-functional characteristics that must be fulfilled by the interface and the implementation in order for a Service to be realized. At this point the Service candidate has been justified for realization and is linked to the complete set of requirements that need to be considered as part of the Service contract.

Figure 4–2 Service Candidate Realization



The justified Service candidate provides the input to Service definition. This process identifies Service boundaries, constructs the contracts from the separated requirements, attaches any enterprise specified contracts, and performs a feasibility check to ensure that for each Service defined, the technology and resources available will be able to meet the requirements of the contract.

The outcome of Service definition is one or more Service contracts. The contracts feed into the Service design procedures, but also enable Service testing to begin the preparation necessary to later test the completed Service. Once the Service contract has

been defined, the test criteria for the Service can be created by analyzing the Service contract.

Service Definition represents a transition phase within the Service lifecycle as it defines one or more Service contracts from the Service candidate. The Service candidate is included as part of the Service lifecycle, because its sole purpose it to justify requirements for realization as shared Services, and is therefore directly responsible for the definition of one or more Services. The Service candidate also provides the mechanism to track a Service's contract all the way back to the requirements. This connectivity among information (hosted within an Enterprise Repository) will serve as a valuable tool when analysis is done to determine SOA effectiveness, and other useful statistics.

4.1.1 Influencing Factors of Service Definition

Typically, a Service candidate will result in a single contract being defined which leads to a single Service being realized after Service delivery has completed. However, a key component of the Service definition process, determines if one or more Service boundaries are necessary in order to make the best use of hardware resources and satisfy the intended consumers requiring the functions of the Service candidate.

Therefore, a Service candidate may result in the definition of multiple Service contracts, as there are additional factors to consider when determining the Service boundaries based on a Service candidate's attached requirements. Influencing factors that affect Service boundaries include:

- Varying Scope of Requirements (Multi-Enterprise, Enterprise, LOB, Application)
- Varying Security Policies
- Message Exchange Patterns
- QOS Requirements

It is important to note, that the Service Identification process is used to identify and justify requirements that need to be realized as shared Services. Service definition represents the first, technically focused engineering activity within the Service engineering framework. It is true that the Service contract is stated in business terms; however, engineering disciplines are now required to ensure that the function of the contract can be realized technically.

4.1.2 Performing Service Definition

The Service Definition process consists of two major stages.

- **Boundary Analysis** - The first stage identifies the Service boundaries by analyzing various influencing conditions against the Service candidate. Services are defined along Service boundaries.
- **Contract Definition** - Once the Service boundaries have been identified, the second stage defines a Service contract for each of the separate Services identified by doing the boundary analysis. This stage involves the introduction of the Service contract document, attaching requirements, augmenting the contract with externally applied contracts (or policies) through contract inheritance (optional), documenting all of the characteristics of the Service which must be fulfilled through the interface and implementation.

The Service Contract defines the Service. It provides the concise definition of what function(s) the Service performs, the operating constraints of the Service (such as QOS requirements), security requirements, as well as any Service enablement requirements

such as SLA enforcement, exception handling, logging, etc. Once Service Definition completes, the resulting output is one or more Service contracts which have been defined along Service boundaries specifying the contractual details for a Service.

Figure 4–3 Service Contract Template

The purpose of the Service contract is not to simply state the prescribed function and scope of the Service, but also specify all characteristics that need to be addressed when making the Service available within a shared environment. The other side, or soft side, of the Service contract, consists of details around lifecycle management policies, expected consumer interaction policies, QOS expectations, as well as Service enablement requirements.

Lifecycle management provides definitions for policies regarding release frequency, versioning, deprecation, and funding. These characteristics are not necessarily important to the delivery of the service, but are important when considering maintenance and management of the Service once in production.

The expected consumer interaction policies include the definition of the intended usage of the Service by its consumers. These policies include the definition or description of the expected message exchange patterns, as well as the ability to support compensating transactions.

QOS expectations describe the various non-functional requirements that need to be supported through the Service's implementation that may not be specifically accounted for through the connected requirements. QOS needs to be closely examined and defined for every Service to ensure that it is designed, implemented and deployed appropriately. The omission of defining QOS requirements up front is a typical fault which quite often results in the failure of software projects.

Service enablement policies include all relevant requirements regarding the enabling of the Service within a shared environment. These policies define whether the Service

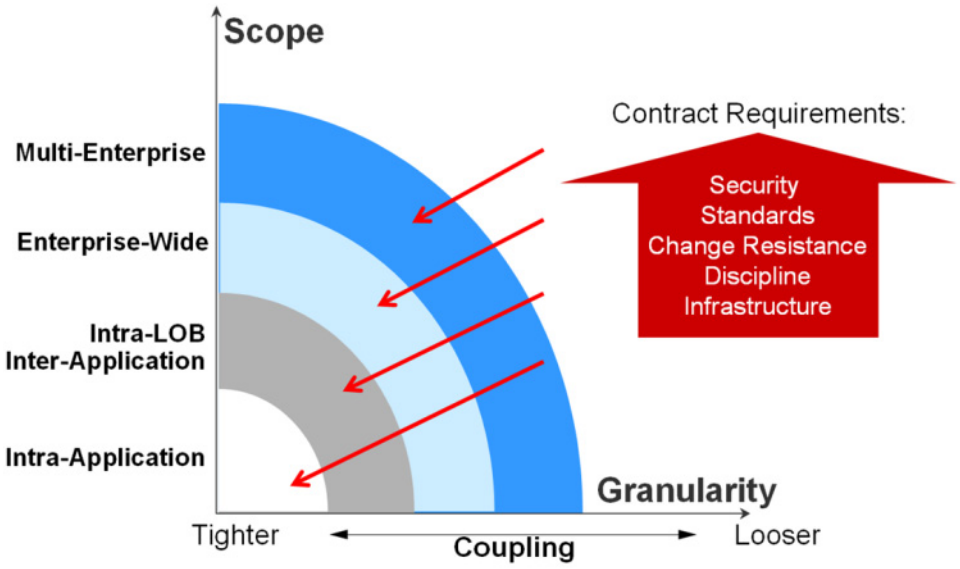
will have managed and monitored SLAs; how exception cases will be handled by the service infrastructure involved in enabling the Service; how the Service will be managed; what level of logging will be required by the Service; etc.

The explicit definition of the soft side of the Service contract is an important enabler for Service design (and implementation), as the Service interface may be significantly impacted by the values provided in these areas from the Service contract. These values also influence provisioning requirements and the deployment structure for the resulting Service.

4.1.3 The Impact of Scope on Service Definition

The scope of a Service describes the breadth of consumers that the Service will be available to serve. The scope of a Service has an even impact across all of the activities of Service definition.

Figure 4–4 Service Scope Influences on Service Definition



Larger scope requires greater upfront investment in developing the Service as more time must be spent designing the Service contract (and interface) to ensure request and response messages are appropriate, standards based (where applicable), and understandable by the larger audience. Increasing scope also places larger demands on defining security policies, as well as infrastructure requirements. Further, any change to the Service contract (or interface) will have wider impact, thus requiring a greater degree of discipline.

4.2 Service Design

Service Design is responsible for crafting the interface for a Service. Although this sounds like a relatively straight forward task, the goal of Service design is to come up with an interface that will maximize reuse while conforming to the demands of the Service contract. The Service interface must be designed with the consumers' needs in mind, but also with an eye towards the non-functional and service enablement requirements dictated through the Service contract. Therefore, Service design may

include a series of trade-offs where a balanced interface between consumer and contract are sought after.

The process begins with the Service contract which is defined through Service definition. The Service contract provides most of the information necessary to develop an interface which is geared towards fulfilling the contract. The second influencing factor of Service design is the consumer. Analysis with respect to how consumers (and potential consumers) will utilize the Service must also be considered in order to achieve a balance between maximizing Service reusability and convenience for the consumers, but also an interface that meets the demands of the contract.

The resulting Service interface is also the final piece of information needed prior developing the test clients that will be used to test the conformance to the contract through the interface, and its implementation. The development of the test cases can take place in parallel with Service implementation.

There is a common misconception that the Service interface consists entirely of the physical artifacts which are deployed to physical systems and exposed through Service infrastructure. This is partly true, in that the end result of Service design produces the physical Service interface, but how did the physical interface come to be? What decisions, considerations and trade-offs were made? What part of the contract is fulfilled by what portion of the interface? The answers to all of these questions (and more) must be documented as part of the Service interface design.

The interface design shares equal importance with the physical interface and it basically shows the work involved by going through the Service design process. The Service design does not need to be a large document, but it does need to articulate the reasoning behind crafting the interface in the manner that it has been crafted. This information becomes especially useful if revisions to the Service are later requested.

Service Design represents the bridge between concept and reality, by tying the contract to the implementation. The Service interface also provides the physical bridge crossed by consumers to gain access to the implementation.

Once Service design completes for a Service, a few things can commence. The first is that the test cases can be created against the Service interface. This allows the test case development to commence in parallel with the Service implementation. Also, projects that are depending on the usage of the Service can now incorporate the interface into the delivery efforts allowing development efforts requiring the Service to proceed. Delivery teams may develop stub implementations for the interface that return crafted results for unit testing purposes.

Another important point to note is that the Service interface is not completely resistant to change (without versioning) until the Service has completed implementation (and testing). It is quite often impossible to completely define a Service interface until all of the implementation details have been worked out (the same is potentially true for the contract also). Therefore, the interface (and possibly contract) may go through a number of iterations as the delivery effort moves forward. Typically these changes are minor and have little impact on dependant applications and test case development. The balance though, is to not get bogged down in design, and simply understand and plan for changes required as the implementation is fleshed out.

4.2.1 Influencing Factors of Service Design

The Service interface is not just a set of operations exposed through a standard protocol and transport. It represents a technical glimpse into how the Service contract will be realized within the SOA.

There are a number of influencing factors that affect the Service design process and the overall crafting of the Service interface. Influencing factors that affect Service design include:

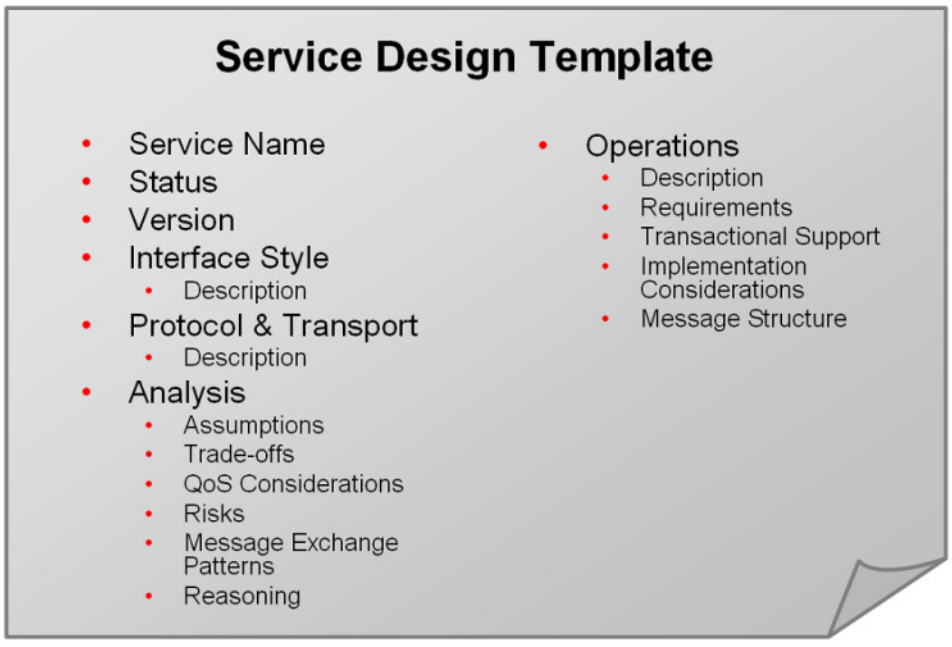
- Scope for which the Service will be accessed
- Service Category
- Functional & Non-functional Requirements
- SOA Service Architecture
- Service Infrastructure

4.2.2 Performing Service Design

Service design begins with the Service contract, and proceeds with the consumer and reuse in mind. When completed, the Service design and interface represent the outputs from the process.

Service design differs significantly from traditional application design, as the focus is not on how the Service will be constructed, but rather, how the consumers will access and interact with the Service. The implementation behind a Service may change several times without impacting the Service design, which is highly unlikely in the case of an application. When a Service design is changed however, the impacts are typically significantly larger than in the application case.

Figure 4–5 Service Design Template



Further, one must consider how certain functional and non-functional requirements are intended to be met. For example, a service might rely on an ESB to enforce security, etc. while another does not. There must be some way to capture this dependency as it has an effect on how a Service needs to be offered. If an ESB is required the interface between ESB and the Service implementation needs to be defined.

The reasons for performing design, whether it is for a Service or an application, are the same however. Design represents a discipline within a larger process that is utilized to ensure both quality, and efficiency. Design also provides the technical reasoning behind the resulting interface, which provides a valuable tool when changes are required to be made to the interface. Interface design is of particular importance to an SOA, and one of the driving goals behind service design is to ensure that the interface created helps enable the SOA, rather than simply satisfy the needs of the known consumers. Quite often the success of any product is determined by its interface. SOA is no different.

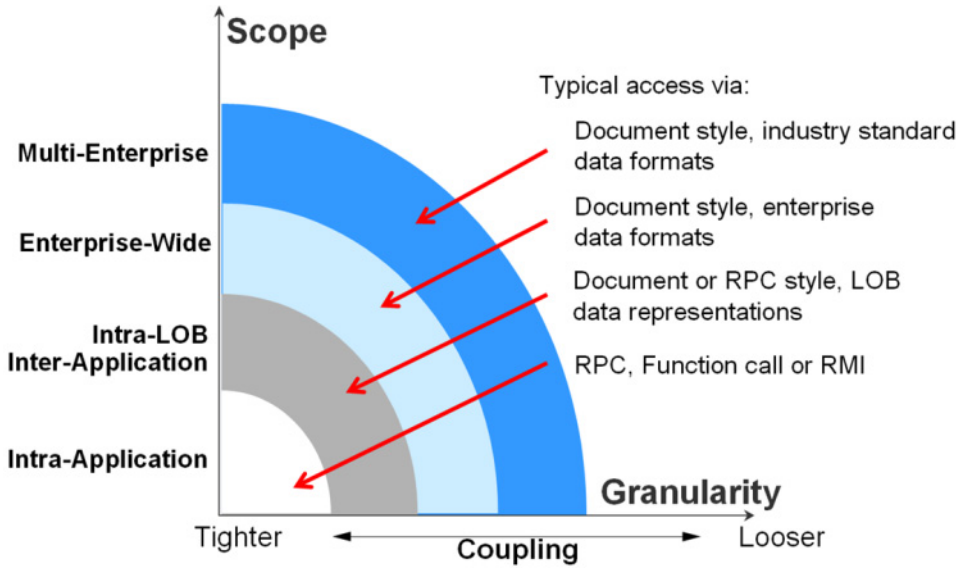
Service interfaces should not be designed for a specific consumer. The interface resulting from Service design needs to be clear, concise and consistent. In order to be consistent, enterprise standards need to exist to ensure consistency. These standards are defined in the SOA Reference Architecture. The crafting of Service interfaces must be considered on the same level as crafting infrastructure interfaces, rather than simply creating a programmatic interface. It is quite possible that several of the Services designed in an SOA will provide an infrastructure-like role (e.g. most utility Services are used to fill infrastructure gaps).

The Service Interface represents the face of the Service. It is the side that all consumers see and interact with. It also provides some of the means for which the contract of a Service is fulfilled and plays a role across all functional and several non-functional requirements. The interface must be designed to fulfill the functional aspects of the Service contract, but should also be designed to enable the non-functional requirements, as well as maximize its potential for reuse.

4.2.3 The Impact of Scope on Service Design

The scope for which the Service will reside in the SOA plays a key role in choosing the style of interface that will be developed for the Service. Consider the illustration below:

Figure 4–6 Service Scope Influences on Service Design



Scope plays a major role in determining the interface style and data formats that will be crafted into the Service interface. As scope increases, so typically, does the number of potential consumers. Larger scoped Services can typically benefit from a document styled interface as these interfaces tend to cut down on the number of requests required to the Service by a single consumer. The trade-off here, or course, is that more data is transported between the Service and the consumer. Document style interfaces also provide a lot of flexibility through Service infrastructure that supports content based routing, as well as data transformations and aggregations.

4.3 Service Implementation

Service Implementation is responsible for delivering the implementation of a Service. This implementation is exposed through the Service interface, but is not only responsible for implementing the interface in accordance to the contract, but rather the entire Service in accordance with the contract. This includes coverage of all functional and non-functional requirements, as well as the Service enablement requirements for the Service.

The "analysis" required to be done prior to beginning Service implementation comes in the form of the work completed in the Service Design process. Augmented with the Service contract, the Service design document and the physical interface provides the necessary information required to begin Service implementation.

Service implementation should really be treated as its own smaller scale software engineering project. The resulting Service Implementation must be passed through Service Testing prior to the completed Service being delivered into production. Much like traditional delivery processes, the development of the test plans, and test cases may take place in parallel with the implementation effort, which will go through a set of design, implementation, and unit test cycles prior to entering formal Service testing.

Service Implementation represents one of the final steps that take place prior to a Service being realized in a production environment. It also serves as a transitional process from Service design through testing and deployment. Service design may be impacted by the activities of Service implementation. It is close to impossible to consider every detail during the Service design phase, and in order to ensure efficiency, some details may be uncovered during service implementation that will influence design. Therefore an iterative element exists between Service design and Service implementation in order to facilitate the handling of any design details that may have fallen through the cracks. If major design details are being missed however, then some enhancements to the design process are required to prevent future recurrences.

Similarly with design, Service implementation also has interaction with the Service testing cycle prior to being released into Service deployment where it becomes operational. For obvious reasons, Service testing will more than likely result in changes to the Service implementation, if not only to fix defects. This may have a ripple effect all the way back to Service design. In severe cases, Service definition may need to be revisited as a result of defects encountered during Service testing, and if so, process improvements are more than likely necessary to prevent future recurrences.

The primary influencing factor of Service Implementation is the Service contract. Secondary factors include the Service design, and consideration for the consumers. This is because the primary goal of service implementation is to fulfill the terms of the contract.

A great deal of discipline is required when engineering and implementing Services. This is because of the strategic importance that these Services provide, as well as the wide variety and number of consumers (and potential consumers) that will be

utilizing the Service. Some Services will have an almost "infrastructure" like purpose within the SOA, and as such, all Services should be implemented with this in mind.

4.3.1 Performing Service Implemetation

One of the most overlooked aspects of Service engineering within an SOA is Service Implementation. It cannot be considered just a simple-matter-of programming which is quite often the case when implementing projects that are not being designed to be reused by a potentially large population of consumers.

Service Implementation is much more than simply writing some code that implements the interface provided through Service design. In reality, service implementation has some parallels to delivering a small, traditional software engineering project that picks up after the analysis phase (analysis is pretty much covered within Service design). In fact it is a good practice to utilize a streamlined version of a traditional software engineering methodology for the purpose of Service implementation. For the most part, a simplified design, implementation, and unit test cycle will suffice, regardless of the method used; the key is to be consistent.

One of the goals behind SOA is to enable flexibility and reusability within the enterprise. This same goal needs to be examined from within the realm of Service implementation. A lot of the infrastructure designed for Service enablement is geared towards making Service implementations more flexible. Further, these technologies often allow for Services to be customized at runtime. Routing technologies provide a good example. There needs to be a governance model around how changes are made within the runtime environment, but it is a good practice to utilize these capabilities whenever possible.

In the end, infrastructure used in this manner increases flexibility, and business agility. It also cuts down on the time and effort required to make the changes in code, the risk in deploying new changes to previously stable code, as well as the potential downtime required for the deployment.

4.3.2 Service Dependencies

There will be many instances where a Service implementation could benefit by reusing the capabilities of another Service. In this case the Service implementation is a potential consumer to the other Sevice. If this usage was not anticipated through the Service identification & discovery process, or even Service definition, then the reuse justification would be required to reuse the Service in its Service enabled form. In some cases though, the overhead required to invoke the Service through Service infrastructure (such as a web service), may require more overhead than the non-functional requirements in the contract will allow. In these cases there are a couple options.

The first option is to invoke the functions of the service using an alternate technology that is better performing such as RMI. In this case you are still doing runtime reuse, and justification would be required to reuse the service in this manner. Significant runtime performance improvement may be possible however. The drawback is that you are bypassing the Service enablement components of the Service you are dependant on. This means that any benefits gained by using infrastructure may be lost, such as statistic generation, audit, etc. In fact your service may be skewing historical results gathered. These conditions need to be considered by the SOA Leadership team prior to justification of the requested usage of the Service in a non-standard way. If this type of reuse will be allowed, the dependency between the two Services still needs to be recorded in the enterprise repository.

The other option is to create a component library that will be used by both Service implementations. These components consist of the various functional elements of the implementation that are relevant to both Services. In this case design-time reuse is being used, which is less restrictive. This option has the advantage of being the most optimal from a performance perspective; the burden is then placed on the maintenance side, since the code has to be managed in a shareable source code repository, as well as any configuration requirements will have be mirrored for each deployment of the library. The code for the library still needs to be branched when any changes are required to the library. In this scenario the two services do not have a dependency between them as they may still evolve independently.

4.4 Service Testing

Service Testing begins after service definition and continues throughout Service delivery (and beyond). Once Service definition has completed, the set of Service contracts have been defined which provides the necessary test criteria required before Service test planning can begin. Service test planning takes place at the same time as Service design. Once service Sesign has completed, the interface can be used to construct the Service test cases, which should be constructed in parallel with Service implementation. The actual testing cycles proceed once service implementation has advanced far enough to begin functional testing. Service testing, with respect to Service delivery, concludes once all acceptance criteria have been satisfied.

Service Testing, when done effectively, provides the glue necessary to generate a quality Service through the Service Delivery processes. It ensures that discipline is used to deliver Services and provides the necessary validation required to ensure the contract has been realized through service delivery prior to the Service entering operational state through Service deployment.

Service testing does not conclude with the delivery of the Service into production however. Regression testing will be required throughout the operational life of a Service. Further, the complete line of test assets for a Service will be revisited on any future versions resulting from approved changes to the Service.

There are two primary purposes behind Service testing. The first is the obvious, which is to ensure that the completed Service realizes the terms of the contract prior to deployment. The Second purpose is to gather the necessary information to determine how the Service will operate within its physical environment. This information is then used to determine the optimal deployment strategy for the Service

Service testing also serves as the gates that must be passed before a Service is allowed to proceed with production deployment, and in this regard is a governance structure. The purpose of Service testing with respect to being a governance model is not only to ensure quality and consistency across Service deliveries, but also to protect the operational environment into which Services will be deployed into.

Iterations with Service implementation may take place after each test cycle, and may even be required after production deployment if defects are found while a Service is in its operational state (which would require an increment to the Service version).

In the case of a new Services entering Service deployment, Service testing provides the necessary information needed to make sound deployment decisions such as packaging and optimal configuration.

Service testing represents the last remaining hurdle for a Service before becoming operational through deployment. The execution of Service test cases begin once implementation has proceeded far enough to have some capabilities through unit test and in a state ready to begin more rigorous functional testing. Test execution and

service implementation will typically go through several iterations prior to concluding with the acceptance being satisfactorily achieved.

Since good Service design follows the practice of design by contract, it is also appropriate to apply test driven development practices within enterprise Service engineering. Practicing test driven development means the test developer builds the test suite independent of constructing the code that implements the Service. If the same developer responsible for service implementation is also responsible for testing, then the test cases should be constructed prior to developing the code. This forces the test developer to build a test suite against the Service contract instead of ending up with the typical "works as implemented" test.

4.4.1 Test by Contract

Once Service definition has been completed the Service contract will be available. At this stage, Service testing may commence for that Service as the test criteria is available in the form of the Service contract. This information allows for the creation of the test plan which needs to provide coverage for all of the following areas within the Service contract:

- Functional Requirements
- QoS (Non-functional Requirements)
- Service Enablement

Constructing the test plan independent of the Service design and Service implementation is a key enabler of the "Test by Contract" concept. This concept requires that the test team understand the contract within its business context, and formulate a test strategy around validating the requirements of the contract. This contrasts with many testing methods where the test team constructs their testing strategy by having conversations with the delivery team, which results in testing against the delivery team's interpretation of the contract, which has never been a goal of testing; however in many cases it seems to be the reality.

Determining the acceptance criteria is pretty simple. It is the Service contract. In some cases it may not be completely possible to realize the complete contract as it is stated, at least within a reasonable timeframe anyway. In these scenarios changes to the contract must be justified and approved by the SOA Leadership team in conjunction with the business in order to achieve acceptance. If agreement cannot be reached, then additional work must be done to realize the contract as stated, or the Service must be abandoned.

In the event that the Service is abandoned, then the dependant projects will become responsible for implementing the functions they required from the Service as part of their delivery effort. In these cases, the implementation assets should be transferred to the ownership of the project delivery teams. If this scenario becomes common, then the Service identification process will need to be reworked with more stringent justification criteria to prevent unrealistic Services from entering Service release planning and delivery.

4.4.2 Test Deployment Options

One other function that should be performed is the testing of various deployment options. The packaging of Services and the co-deployment with other Services and dependencies may lead to a few deployment options that need to be tested in order to determine the best option. If analysis leaves room for doubt don't be afraid to run a few tests to determine the optimal deployment packaging structure.

5

Service Management

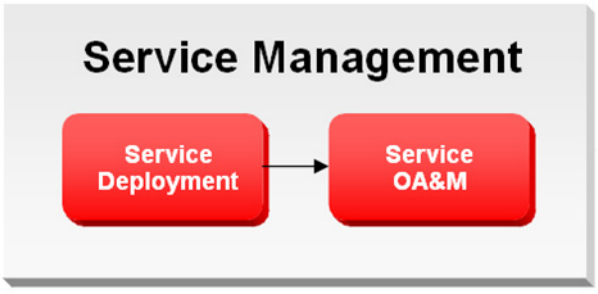
Service Management is a key SOA enabler, and involves much more than simply deploying and maintaining the Service within its operational environment. Service Management when done correctly facilitates a number of important objectives of SOA, such as:

- Reuse
- Flexibility
- Service Evolution
- SOA Measurements (Success, reuse, adoption, conformance, etc.)

Obviously, Service Management does not bear sole responsibility for the objectives listed above, but it can certainly affect their effectiveness and if done improperly, hinder the overall effectiveness of the SOA.

Service Management is a much larger topic then is covered by this document. This section of the document highlights the areas of Service Management that an architect and developer are concerned wih. For more details on Service Management see the *Service Management in a Fusion Enviornment* document.

Figure 5–1 Service Management Phase



Service Management represents a transition phase and is influenced directly by information gathered as far back as Service Release Planning, where resources and hardware provisioning is initially planned. Service Management covers the Service through deployment and how it is administered, operated and managed in the operational environment. This also includes enabling Service Provisioning where it is made available for reuse by subsequent projects.

There are both design-time and run-time aspects to Service Management. From a design-time perspective, Service Management affects the way in which Services are discovered, measured, analyzed, and evolved.

Technology also plays a major role in this area, but more importantly, the way the technology is applied is of upmost importance. It is completely unrealistic for an enterprise to attempt to adopt SOA by building their entire infrastructure in-house. The support for standards within Service based technology has been maturing over the past several years, which has allowed infrastructure to be used as not only a hosting environment for Services but also provides flexible Service Enablement functionality.

Service Management represents the final states of a Service in its lifecycle. Once operational a Service remains in that state until decommissioned, or it evolves through Service versioning. The magnitude of the change required to the Service for the new version dictates the states that the new version must pass through prior to returning to an operational state. The previous version of the Service remains operational until the new version is deployed. In cases where the new version is not backwards compatible, the previous version of the Service may remain in production after the deployment of the new Service version in order to facilitate the time necessary for all consumers of the previous version to upgrade to the new version.

The goal of Service Management is to enable the SOA to achieve its objectives. As mentioned earlier, the objectives of the SOA are not the complete responsibility of Service Management, however, the policies and procedures employed through Service Management are crucial to overall SOA success. One of the primary goals of Service Management is to maintain control over the SOA assets. Service Management has the responsibility to ensure that the facilities are in place to enforce the guidelines of the SOA. (Refer to *A Framework for SOA Governance* and *Service Management in a Fusion Environment* document for more details)

If Services were simply published for reuse after being delivered through Service delivery, and then deployed on any available infrastructure with no control over Service consumption the SOA would rapidly become ineffective. One very popular Service could end up bringing down the entire environment due to overuse or misuse. Further, the cause of the problem may be very difficult to manage. It is true that Services are designed to be shared, but how Services are made available and what consumers will have access to Services needs to be controlled and measured. This will enable the SOA to grow and evolve in a manageable fashion.

The rules and guidelines established around Service Management will vary from enterprise to enterprise much the same way the objectives of the SOA vary. Some enterprises will have stricter requirements in certain areas and looser requirements in others, these requirements need to be based on the specific enterprise's business needs.

Service Management affects reuse, as it provides the procedures for exposing Services for discovery and participates in the process for reuse authorization. Services Management is responsible for tracking Service inventory, and providing a system for allowing the natural discovery of Services from within the Service Identification & Discovery process. Service Management also tracks Service dependencies and consumptions rates, and is responsible for taking a proactive approach to environment growth based on Service usage statistics.

Service Deployment strategy plays a key role in enabling flexibility. The packing of Service infrastructure, applications and Service deployment units is a crucial element of SOA flexibility. Having a strategy for deployment of the various participants within the SOA is required to ensure that maximum flexibility can be achieved while minimizing trade-offs such as performance, and manageability.

Service Management also plays a role with respect to the ability for Services to evolve. Service Management is responsible for managing Service versions and the deployment and provisioning of new versions. The environment needs to be established in a way that supports Service versioning and even the parallel deployment of multiple Service versions.

5.1 Service Deployment

The shared nature of SOA demands a new strategy when it comes to deploying components of the SOA. Traditionally, applications have been deployed and managed autonomously in their own technical silos

A lot of enterprises have been building "Services" as part of their application delivery efforts, which are then bundled up and packaged with the application and deployed to the same processes on the same servers. This strategy poses a couple major problems:

- The Service cannot evolve independently of the application.
- If other applications are to consume the Service, they become dependant on the Service's hosting application and not just the Service itself.
- The strategy results in a rigidness that is counter-productive to the goals of the SOA.

With SOA, applications are still deployed in a similar manner, where they have their own lifecycle. However, they are no longer autonomous. SOA applications rely on shared Services which are not packaged with the application(s) they were created to serve. Services therefore, have separate lifecycles. Independent Service lifecycles is a fundamental component of successful SOA and it must be provided for by the Service Deployment strategy. The only time a Service should be allowed to be packaged with its consuming application is when it fails the Service justification process, and is left as the responsibility of the project delivery team. In this case the Service is considered to be an "Application" scoped Service. If the Service becomes justified in a later project, it would need to be moved, or decoupled from the hosting application in order to be realized as a shared Service and take on a scope higher than "Application".

5.1.1 Influencing Factors of Service Deployment

There are several factors that emanate from the service engineering process that come into play when determining the optimal deployment strategy for a particular Service. Influencing factors that affect Service deployment include:

- Service Scope
- QoS Requirements
- Security
- Service Enablement Requirements

Each of the factors defined above provides a single angle of influence with respect to deployment strategy. These influencing factors need to be considered as a whole in order to determine the optimal deployment configuration for a particular Service. Further, deployments are typically a game of trade-offs. Certain benefits gained by any particular deployment strategy results in something lost. For instance, a highly-distributed deployment is very flexible and highly available, but comes at the price of lower performance, and higher maintenance. Therefore, the balance always needs to be weighted. A successful SOA will introduce a variety of deployment strategies which should be applied to different types of Service based on the above criteria. In larger SOA deployments it may be justifiable to have multiple Service

hosting environments tailored through deployment to meet the various combinations of influencing factors.

A Service's scope (defined against the expected consumers for a Service), plays a significant role in determining Service deployment. In general wider scoped (e.g.: enterprise or multi-enterprise Services) consumer bases favor a more distributed deployment structure. Services with a wider scope need to be managed autonomously. Changes to wider scoped Services and other SOA participants at the same level, need to have a minimal effect on one another because of the large number of dependencies, or their strategic importance.

For the most part QOS influences are pretty obvious. Reliability, availability, scalability, and performance all have direct correlations to deployment strategy. A balance is also required with QOS requirements, as an improvement in one value, usually comes at the degradation of another. For instance, availability usually comes at the cost of some performance because of the overhead involved. Reliability can also be negatively impacted by availability and scalability.

Security requirements have a direct impact on deployment strategy. In a properly secured environment, security is configured through hardware and software infrastructure, and is not left in the hands of the developers. Increasing security requirements also come at a cost of performance and possibly even reliability (propagation/synchronization issues, etc).

Service Enablement is much more than simply providing a mechanism to expose a Service to it's consumers through a standards based mechanism (e.g.: web services stack). Service Enablement can also be used to enhance SOA flexibility and support a small portion of the overall service's implementation logic. Service Enablement is achieved through Service infrastructure, which may be configured to expose Services, and ease the difficulty of implementing services. Different enablement requirements may require different infrastructure, which directly impacts deployment strategy. It is a good practice for an enterprise to standardize on a particular set of Service infrastructure components that meet their common Service enablement needs

5.1.2 Service Migration

When Services are identified that are already hosted within an existing application, such as application Services, it is tempting to leave the Service in place within the application, and have new consumers interact with the Service while it remains bundled and co-deployed with the application. There are a number of fundamental problems that arise from this however. Some of these problems include:

- The application maintenance or evolution will require Service downtime, affecting other consumers.
- Changes to the Service from outside requirements will require application downtime.
- The Service is governed by the application, rather than by the SOA.
- The application and Services cannot be independently managed.
- The application and Service(s) cannot be independently scaled.

At the root of these problems is the fact that the application and Services do not have independent lifecycles. At a very minimum, Services and applications must have independent lifecycles. At a minimum, if a Service is to be promoted from an application based Service to a higher scoped Service where it can be utilized outside of the scope of the application it must be migrated to the shared Service environment. Migration is typically not simply moving the software components from the

application and redeploying them in the appropriate shared Service environment. More often than not, additional engineering discipline will be required in order to meet the demands made of the Service by its wider consumer base.

5.1.3 Deployment Unit

When it comes time to deploy a Service to the Service infrastructure a necessary decision needs to be made about what belongs in a single deployment unit. Putting all the shared Services in a single deployment unit is totally inappropriate since a change to one Service will require a redeployment of all Services. Similarly, deploying shared Services as part of a larger application will lead to lifecycle issues since the Services must have a lifecycle independent from the application lifecycle.

The smallest deployment unit would be a single Service with all its required pieces and parts. It may also make sense to create a deployment unit that contains several closely related Services, for example, Services that all use the same underlying database schema or access a common backend system.

Whether a single Service or a collection of Services, the deployment unit must remain autonomous and isolated from applications and other Services. This allows the Service or collection of Services to be administered and versioned independently. This independence is a key enabler of agility.

5.1.4 Service Enablement

Service Enablement is considered a component of the Service's implementation. The difference between enablement and the rest of the Service's implementation is that Service enablement is accomplished through infrastructure; versus deployment based artifacts originating from code (think development tools here). Further, the purpose of Service enablement is to provide Service level QoS for a Service which do not typically stem from business requirements. This means that Service enablement plays a major role in Service deployment. With that point in mind, Service enablement provides the following:

- 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

5.1.5 Service Provisioning

Provisioning is done to make a Service available for consumption, in addition to enabling Service discovery. Once a Service has been deployed and published it becomes available to be provisioned for consumption by approved consumers.

5.2 Service OA&M

OA&M (Operations, Administration & Maintenance) with respect to SOA, is a very extensive subject. In traditional project or application based environments, OA&M had the sole responsibility of keeping the environment operational, administering changes, and handing maintenance. All of these activities had a technical focus and other than keeping the backbone of the enterprise applications operational, provided little additional business value. Service OA&M goes beyond the traditional approach, and has a large focus on harvesting business benefit from the information gathered from the SOA.

In order to enable business value through OA&M, Services need to be engineered correctly to enable Service Management to capitalize on the information made available during the normal operations of the service environment. Of course, the missing link is what information is going to be recorded, and what business benefits will this information bring? This analysis needs to be done prior to beginning Service engineering, otherwise Services may be developed and deployed and the SOA will lack the ability to harvest any additional business benefits through the operations of the SOA.

5.2.1 Service Consumption Requests

Once services have been deployed and published, they should be in a state where they may be consumed by producers. Service consumption needs to be controlled, otherwise there is no way to manage resource utilization and establish any form of control over the SOA. If service consumption is not controlled there is a good chance that the value promised by SOA will deteriorate. The scope of a service obviously affects the control strategy.

Controlling Service consumption from an internal perspective means that consumers can not simply discover and begin to utilize a Service without going through a controlled approval process. In these scenarios a more relaxed approach can usually be taken, where procedures are constructed, and internal consumers are expected to follow them. Consumption is monitored and violations are brought up and taken care of in a reactive nature.

With respect to freely available, externally facing Services, control is established by throttling the environment, and ensuring minimal impact to the rest of the IT infrastructure will result when the external nature of the environment is exploited. The honor system will simply not work here, and a proactive system must be established to ensure security and exploits cannot take place.

As for externally facing services that are not freely available, control must still be established through a proactive method, but additional security requirements must be established to ensure that only trusted and approved consumers are able to access the appropriate Services.

Service OA&M introduces requirements on Service engineering that effect the entire process. The application of the Oracle Service Engineering Framework should not commence until all aspects have at least been considered. The decisions made, regarding the procedures for operating, administering, and maintaining the SOA will introduce requirements that need to be introduced at the beginning of the process. If ignored, the retrofitting of these concepts will be a much bigger burden than if they were at least considered and designed prior to commencing.

6

Summary

Many enterprises have discovered that traditional delivery methodologies tend not to consider the needs and requirements of projects outside of the project in focus. These methods are much too narrowly focused and need to be adjusted to enable SOA, otherwise this leads to major inhibitors to reuse and agility.

These enterprises require an update to their existing delivery methodologies with key Service engineering & modeling activities that will allow for consistency across deliveries for projects that need to coexist.

An enterprise class Service engineering and modeling framework assists with delivering projects within an SOA environment, by not only adding key activities at the project level but also adding key activities such as Service identification at the program level.

The Oracle Service Engineering Framework is an engineering approach for delivering projects within an SOA environment. It identifies and resolves the unique challenges encountered by enterprises adopting SOA. It complements traditional delivery methodologies by defining the engineering disciplines required for effective and consistent service delivery.

A

Further Reading

The ***IT Strategies From Oracle*** series contains a number of documents that offer insight and guidance on many aspects of technology. In particular, the following documents pertaining to Software Engineering in an SOA Environment may be of interest:

ORA SOA Foundation - Provides a description of the foundational aspects of SOA in support of the broader Oracle Reference Architecture. This document is intended to provide historical, as well as current, context for SOA so the reader will understand SOA fundamentals underpinning the ORA.

ORA SOA Infrastructure - Provides a enumeration of the key capabilities required for SOA implementations and organizes them into logical architectural components. Oracle Fusion Middleware products are mapped to the logical architecture and various views of the service infrastructure including physical, deployment, process and development views are elaborated in detail. This document provides an understanding of the best way to implement an effective SOA infrastructure.

ORA Integration - Provides a description of the concept of service-oriented integration and how this differs from more traditional integration approaches. The principles that are essential to a service-oriented approach to integration are listed and the principles are linked to the layers and capabilities included in the architecture.

Refer to the ***ORA Glossary*** document for descriptions of key terms.