

Version Management Process For SOA Services



Integrating the Enterprise

Bill Clarke Consulting

www.BillClarke.com

Introduction	3
What Is Versioned: Using the registry/ repository to manage versions of services	3
Service Versioning and Endpoint Abstraction	13
Governance Policies for Managing Versions of Services	14
1. General Policies	14
2. Policies for Major Releases	16
3. Policies for Minor Releases	17
4. Policies for Revision Releases.....	18
Summary	18

Introduction

SOA services are enterprise wide assets that overtime will have a life of their own independent of the applications that consume the services. They will have feature upgrades, bug fixes, and performance and compatibility enhancements. This means that enterprise services should have version releases just as any commercial software product has version releases.

Unlike most commercial products however, enterprise services will be embedded in consuming applications. The release of a new version of a service cannot disrupt the performance or operation of existing applications that consume the service. Service consumers will need to know what to expect from a new release of a service and how to prepare for it.

This document looks at aspects of versioning services in production and proposes several policies and approaches for enterprise. Specifically it proposes:

- Using the registry/ repository to manage versions of services;
- Using endpoint abstraction to support runtime management of service versions;
- A version numbering scheme based on common industry practices;
- A set of governance policies for managing versions of enterprise services.

What Is Versioned: Using the registry/ repository to manage versions of services

There are two main components of a service: the contract, and the implementation. Services are functionality required by a set of consumers. This functionality is the implementation of the service – often referred to as the service endpoint . A service consumer needs to understand how to access the service endpoint along with any constraints the service may have. The service contract is used to communicate to the consumer what the service does, and how to use the service.

The Service Contract: The service contract for a service is one of the most important artifacts created by the service developers.

The service contract specifies the purpose, functionality, constraints, and usage of a service. In effect the contract can be viewed as “Service’s Rules of Engagement” in a language that both the consumer and provider understand. The services contract has two parts:

1. The service contract document and;
2. The Web Services Description Language file (WSDL)

Both of these artifacts are stored in the registry/repository and are discoverable by searching on the taxonomy metadata that tags the artifacts.

The service contract document is essentially the user documentation for the service. It documents how the service is used, its functions, its interfaces (input data, output data), runtime policies, capacity and response times (SLA and QoS), availability, data quality, etc. AgilePath advocates a “Contract First” development model. The services contract should be one of the first specifications developed as part of the development lifecycle. The service contract should be entered into the repository as a discoverable artifact very early in the development lifecycle – as early as possible. This allows potential consumers to understand what they can expect from a service when it becomes available and they can design their consuming applications accordingly.

A service contract can be with all consumers, or with specific consumers. If the contract is with all consumers, it describes how to use the service as well as the functions and SLA/SLO that all users can expect. However certain users may have unusual or unique requirements – for performance, availability or specific versions of an implementation. When this is the case the service contract will specify an ‘agreement’ between the service provider and specific consumers of the service, even if these consumers are all using the same version release level of the service.

The Service Contract Document

We propose 4 sections to the service contract document:

1. Service Functional Profile:

This section is not a full technical description, but includes only the details required to understand the functionality of the service, including the operations (methods) and message (data) types. This section also includes references and taxonomy information about the service.

2. Service Governance Profile:

This section includes details related to governance, such as SLA policies, security and authorization requirements, transactional or reliable messaging policies, etc; This section includes policies intended for runtime execution, monitoring, or enforcement, but also policies intended for human interaction or behavior, including service lifecycle behaviors and approvals, and documentation of sponsors, approvers, custodians, etc.

3. Abstract Service Contract:

This section includes all functional or governance elements for which there is support for WSDL 1.1, WS Policy, WS Policy Attachment, or other WS* standards implemented in XML metadata. The main difference between this section and the full “Implemented Service Contract” is that the Abstract Service Contract includes specific endpoint information, and may not include transport information; this allows the Abstract Service Contract to be portable across implementation environments.

4. Implemented Service Contract:

This section includes all information about the endpoint information fully included; i.e., this level supports actual runtime execution. To the extent that the implemented systems support dynamic or other managed configuration, this contract level may not need to be explicitly or fully realized as a persisted document.

The Web Services Description Language file (WSDL)

The Web Service Description Language file (WSDL) is a part of the service contract. It is the technical specification for accessing the service. The WSDL governs the creation and management of network endpoints, or ports for the service. The WSDL takes advantage of a key abstraction of the service’s implementation access points and messages. In the simplest description, a WSDL describes the public interface to the web service.

In a “Contract First” development model the WSDL should be inserted into the registry as soon as practical. This allows potential consumers use the WSDL as they design and implement consuming applications, even before the actual service implementation is completed.

The WSDL will reference and use XML Schemas and WS-Policies. These are separate artifacts that may be new or existing. In either case they are files that will be versioned along with the WSDL.

Service Implementation

The implementation part of a service is the actual code, application interface or other technology asset containing the functionality needed to realize the service. The service implementation may use any technology, such as Java, C#, Perl, PHP, etc. The service provider is the only one concerned with the details behind the service implementation. The service consumer should have no knowledge of the implementation details, and if it does, you are right back to creating tightly coupled point-to-point systems.

Relationships Among The Artifacts

The registry/repository should store the various specification documents, service contracts and the XML artifacts needed to use and access the service. It should also be able to maintain the relationships among the artifacts while supporting versioning of the artifacts.

Figure 1 shows the logical relationship among the different parts of a service specification and service contract stored in the registry/repository.

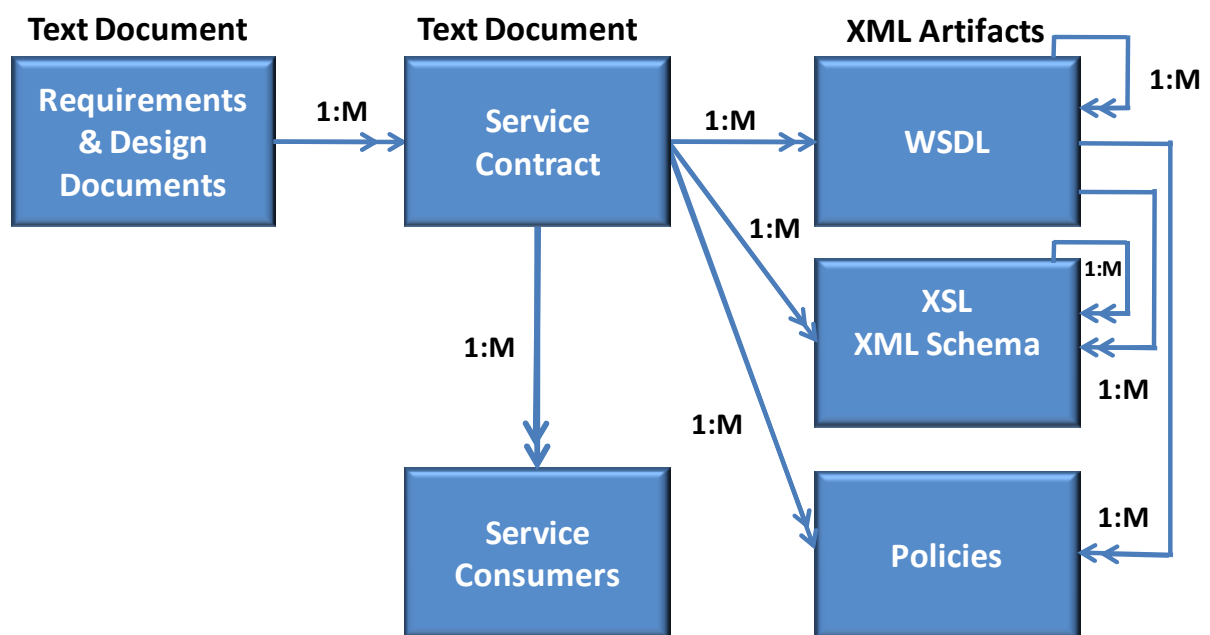


Figure 1
Logical Registry Entry

Let's step through this model. Each of the logical entities in the registry/repository is shown as a rectangle in the diagram. The double headed arrows show how the entities are associated with one another. The annotation next to each connection ("1:M") shows the cardinality of the

relationship, which is the number of entities on each side of the relationship. For example, one-to-one, one-to-many and many-to-one express cardinality. It turns out that the cardinality for each relationship in this diagram is "one to many". That means, for example, that there is one service contract for many service consumers –or said another way, many service consumers can share the same service contract. In some instances the connecting arrows point back to the same entity – for example the WSDL. This means that a WSDL can contain many other WDLs (1:M) and that an XML schema can contain other schemas.

Entity relationship diagrams such as this are extremely useful and concise, and show important business rules. Now let's look at each one of these logical relationships.

Requirements & Design Documents.

The registry/repository can be used to store versions of high level service design documents with the final approved version associated with the service contract document (described earlier). Including design documents in the repository is optional, but it is recommended by AgilePath.

Service Contract and Service Consumers

The service contract is a document that describes what potential consumers of the service need to know to access and use the service. This document was described earlier (see page 3). It is essentially an agreement between the service provider (or service owner) and the consumer of the service.

Each consumer is associated with a single service contract. Consumers will typically share a single service contract and share the same version (1:M cardinality) of a service contract. There can be instances when one or more consumers have unique or different requirements and therefore have different service contracts. This may be the case if, for example, the service consumer has very specific QoS or availability requirements that are different from other consumers. So there can also be multiple versions of a service contract.

In any event, all users (consumers) of a service should be registered in the registry/repository and associated with a service contract.

XML Artifacts

Most commercial registries are integrated with a repository. The repository is designed to store and manage documents and can have any taxonomy or organization that fits your needs. The registry however will typically support a standard schema or internal organization. The most common is Universal Description, Discovery and Integration (UDDI).

Here is a logical model for UDDI version 3:

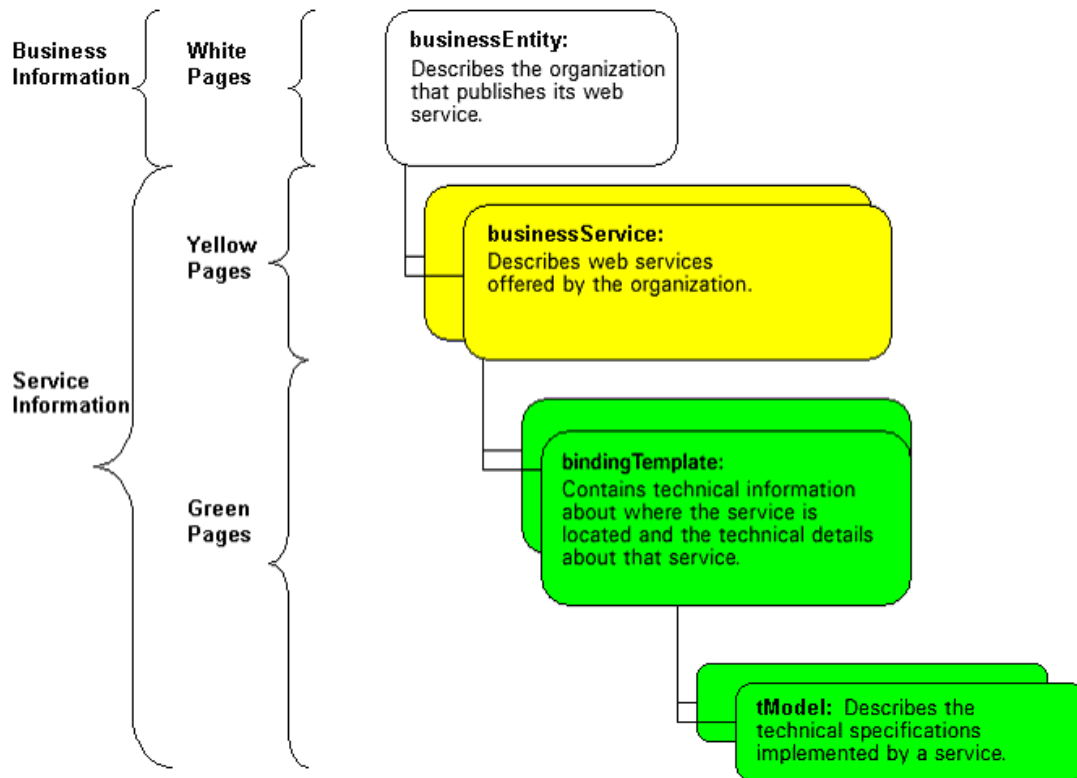


Figure 2
UDDIv3 and Taxonomy

Registry stores and manages the XML artifacts in the 'tModel' entity through a pointer to the XML files. Since most commercial registries are integrated with a repository, the logical entity model described earlier can be implemented across both the registry and the repository.

There are three types of XML artifacts:

1. WSDL files
2. XML schemas
3. WS-policies

WSDL files are a technical description of the service and how to access it. The file is an XML file

Element	Defines
<portType>	The operations performed by the web service. The <portType> element can be compared to a function library (or a module, or a class) in a traditional programming language.
<message>	The messages used by the web service. The parts can be compared to the parameters of a function call in a traditional programming language.
<types>	The data types used by the web service. Uses XML Schema syntax to define data types.
<binding>	The communication protocols used by the web service. The <binding> element defines the message format and protocol details for each port.

Figure 3
Elements of a WSDL file

used to specify the operations, messages data types and communication protocols for the service. As shown in the previous logical entity diagram (see page 3) the WSDLs can contain other WSDLs (for compound services), multiple XML schema and multiple policies enforced at runtime.

Each of these artifacts can be versioned separately, but should be related as a complete configuration – similar to a ‘build’ in a software configuration management system.

A Version Numbering Scheme

One approach to service versioning is to work with a release numbering system that uses the concepts of MAJOR, MINOR, and REVISION releases. Here is one very common structure for these numbers:

major.minor.revision

Major: Major releases demarcate significant changes to a service. Service providers must perform a major release whenever the new release is not interface-compatible with the previous release. Also, service providers may perform a major release if there have been substantial improvements to the service. A major release is usually referred to as a ‘dot-zero’ release and typically has the following characteristics:

- Not backward compatible – either an interface and/or input/output message types have changed.

- Deprecated operations have been removed.
- Can be used if service has gone through a major rewrite or been updated considerably.

Try to avoid or minimize the number of these release types. If this release type is common it tends to indicate that the current “Service Identification” process is not optimal and may need to be reviewed. Since major releases are not necessarily backward compatible with previous releases, all consumers will need be notified well in advance and given options if they cannot upgrade to the next release. More will be said on this topic in the ‘policies’ section of this document.

Minor: Minor releases signify enhancements to a service that do not necessitate a major release. Service providers may perform a minor release if the release is interface-compatible with the previous release; i.e. 1.2.0 will be backward compatible with 1.1.0. A minor release is usually referred to as a ‘dot-X’ release and has these characteristics:

- Backward compatible.
- New operations allowed.
- New data types allowed.
- Addition of optional fields allowed.

The interface signature remains the same, but its behavior or a quality has changed, to comply with the changed specification; although the consuming application still works, the consumer may still want to consider whether the change in behavior or quality is acceptable and what impact it has. This requires a service modification communication workflow process to be in place.

Revision: A revision typically involves simple bug fixes or optimizations that do not introduce new features. Developers may perform a revision if the revision is 100 percent interface-compatible with the previous release. A revision is usually referred to as a ‘dot-X-dot-Y’ revision such as 2.1.1 and has the following characteristics:

- No new functionality
- Focuses on specific problems such as performance or defect fixes

There may be instances where specific consumers use unique configurations of a service revision, these are describe in the following section of this document. In this case there may be a letter designation appended to the revision number: 2.1.1a, 2.1.1b, 2.1.1c ... etc. These may be used to designate the same version release level for a service, but different endpoints for

specific consumers or groups of consumers, or different run-time policies for different user groups.

Version Dependent Service Consumer Invocation

There are two fairly straightforward ways for the service consumer to direct a request message to a specific service version:

- URI Reference
- Header Information

URI Reference: In a Web Service, the physical location of a service is defined by the <port> tag in the WSDL. In the situation of a Minor release, the physical location will more than likely remain the same, and thus the consumer could continue to reference the existing URI. In the case of a Major release the physical location of the service would be different, and the consumer could just point to the new physical location described in the WSDL.

Header Information: By embedding the version information in a standard SOAP header, a dynamic routing decision could be made at run time. With a policy enforcement proxy or enterprise service bus (ESB) this type of decision-making is very easy to do, and provides significant flexibility in how different versions of services are accessed. This is the preferred method.

A variation on this involves allowing the routing decision to be driven by service-side policy enforced by intermediaries (such as a policy manager or ESB) at run time. Instead of a header from the client identifying a specific version, the a policy manager or ESB can examine the header for information that identifies the consumer, consuming application, or consuming organization. The information can range from the authenticated user to a keyword placed in the header by the client. One advantage with this mechanism is that it is usually used in conjunction with repository features for dependency mapping of the consumer types to the services; the same mechanism used to identify the consumer for routing the request assists in tracking versions of the service are used by which consumers. This enables more proactive notification and impact analysis of changes. There will be more discussion of this type of version management in the registry/repository in the next section of this paper.

The ICD is versioned with Major and Minor release levels (1.0 and 1.1 for example). These release levels would also require new versions of the service contract (including new versions of the XML artifacts). Revision release levels (1.1.1 and 1.1.1a) on the other hand will have no functional changes so only the Service Contract would be versioned (as well as the

corresponding XML artifacts, since they are part of the service contract). Let's walk through some example to illustrate the process.

When a service is first released it's at version 1.0. The service owner would turn over to the enterprise Program Office (the enterprise registry librarian) the Interface Control Document (ICD), Service Contract and XML artifacts for version 1.0. If the next release had major functional changes that are not compatible for the first release, this would be release 2.0 and it would have an Interface Control Document (ICD), Service Contract and XML artifacts for that release.

If the next release had functional changes but was still backward compatible with the current release, it would be a "dot x" or minor release (for example 2.1). This release would also have its corresponding ICD, Service Contract and XML artifacts.

If the next release had no functional changes, but only bug fixes or changes to runtime policies (such as changes to the endpoint), this would be a Revision release. Since it has no functional changes, only the Service Contract and the XML artifacts would be versioned. The text of the new Service Contract would describe the differences and the version number in the WSDL and other XML artifacts would be changed.

If a particular consumer had unique run-time requirements – such as security or dynamic re-direction - this would be spelled out in a separate service contract for that consumer and implemented by corresponding runtime policies that re-direct a service request to the appropriate endpoint implementation. If it is still using the same version of the endpoint , this revision could be labeled with an alpha appended to the revision number (such as 1.1.1a or 1.1.1b, etc.).

Service Versioning and Endpoint Abstraction

Endpoint abstraction means that the endpoint location is not coded into the WSDL as an URI Reference. This means that a service request can be redirected without changes to the WSDL or to service policies. The content-based routing policies for the re-directed service request are guided by information in the SOAP header that identifies the user or the class of user making the request. Different users with different QoS requirements for example, could be redirected to different endpoints in different servers. In the enterprise component architecture this functionality could be handled by the service container.

Endpoint abstraction is very useful for managing versioning and ensuring the appropriate version of a service is available to the appropriate users (consumers).

Service Contract Portability across SDLC Environments

Endpoint abstraction allows the different versions of a service to be available across multiple environments, such as development, QA, and production environments, with more consistent results due to all details short of machine and port names or other transport protocol details being as identical as possible. This contributes greatly to the general SDLC goal of fielding systems across the lifecycle that are as close as possible to the code and configuration of systems tested in previous lifecycle stages. In practice, especially in the case of very complex services clients such as business processes and orchestrations, this can be critical to development and QA efficiency.

Abstraction of Service Implementation for Service Versioning and Change Management Support

For production environments especially, abstraction of the endpoints can enable multiple versions of services to be supported. UDDI queries or content based routing are among the mechanisms that can be used to route messages to the version required by a given consumer. This can contribute greatly towards mitigating and managing the impact of changes, and allow for separate schedules to be fielded for different consumers.

Abstraction of Service Implementation for Physical Provisioning

For production environments especially, abstraction of the endpoints is a key enabler of IT provisioning, allowing server or IT environment changes to be made with little or no impact on clients.

Governance Policies for Managing Versions of Services

This section provides policy guidelines for managing versions.

1. General Policies

1. The version level of a release should be a joint decision of the enterprise program office and the service provider
2. Endpoint abstraction should be used to promote and demote services
3. Service providers should be responsible for maintaining service endpoints

4. The Service Provider should inform the enterprise program office of any changes to the endpoint implementation – either version promotion or demotion or re-direction policies
5. The service contract should be used to specify which version(s) are being used by which consumers (consuming applications)
6. All consumers of a service should be associated with a service contract
7. The enterprise program office should determine if and when a enterprise service is promoted to production and made available for consumption
8. The Registry/repository should contain accurate descriptions of the current release levels of all enterprise services
9. All enterprise service should have one of four states in the Registry/repository: Proposed, Under development, In Production, Not in production.
10. Registry entries for all previous versions of a service should remain in the Registry/repository with a status of “Not in production”.
11. enterprise service providers should be responsible for a accuracy of the service contracts in the Registry/repository
12. enterprise service providers should be responsible for a accuracy and currency of the service documents in the Registry/repository
13. Updated to the enterprise registry should be performed by the enterprise Librarian
14. All users (consumers) should use the same version of a enterprise service, unless specifically granted permission from the enterprise program office
15. The version of a service used by consumers should be described in the service contract
16. All new users of a service should use the most current major version of that service
17. The meta data of a WSDL in the enterprise registry should contain the major and minor version number of the service

18. The endpoints for major and minor releases should be updated such as it will be possible to re-direct a service request to the previous release (this is essentially 'rolling back' a release).
19. Updates to endpoint implementation should follow established FAA versions and configuration control policies.
20. The Registry/repository should be used to manage versioning of IRD, ICD, Service Contracts and XML artifacts for all enterprise services.

2. Policies for Major Releases

1. All version releases that require existing consumers to update their consuming applications should be major version releases (are not backward compatible)
2. All version releases with incompatible signatures (xml schema) should be major releases
3. All version releases that add or deprecate functionality should be major releases
4. Service providers are responsible for contacting existing consumers of a service and the enterprise program office at least 6 months before a major release
5. All service providers should communicate the anticipated impact of a major release to all existing consumers six months before a major release
6. Incompatibilities of a major release with an existing release should be included in ICD (Interface Control Document) in the Registry/repository
7. An existing version release should remain available and operational for six months after a major release
8. Only existing users of a previous version should be allowed to use that version for six months after a major release.
9. All new users of a service should use the most current major version of that service
10. Six months after a major version release all users should be upgraded to the most recent version. This date should be called the mandatory cut-over date.
11. Existing users of a service should be required to request permission to continue using the service for six months after the mandatory cut-over date.

12. The availability date of a Major service will be available in the registry/repository
13. The mandatory cut over date for a major release should be available in the registry/repository
14. All major releases should have new versions of the WSDL
15. The meta data of a WSDL in the Registry/repository should contain the major release number

3. Policies for Minor Releases

1. All minor release will increment should
2. All version releases that do not require existing consumers to update their consuming applications should be minor version releases (are backward compatible)
3. All version releases with backward compatible signatures (xml schema) should be minor releases
4. All version releases that do not add or deprecate functionality should be minor releases
5. All minor releases should use the WSDL file of the previous release
6. All existing users of a service should be notified of a minor release one month before its promotion to production
7. An existing version of a service should remain available to current users for six months after a minor version release
8. Six months after a minor version release all users should be upgraded to the most recent version. This date should be called the mandatory cut-over date.
9. Existing users of a service should be required to request permission to continue using the service for six months after the mandatory cut-over date.
10. All minor releases should have updated ICD documents.
11. All minor releases should have updated service contracts (except for the WSDL file).
12. The service contract for a minor release should contain a description of all changes made to the service in the new release
13. The endpoint for a minor release should

4. Policies for Revision Releases

1. All revisions should be 100 percent compatible with existing releases
2. Revisions should not change any functionality or operations of the existing service release
3. Endpoints should not change with a revision
4. Existing users who explicitly request notification of revisions should be notified of a service revision
5. The service contract for a revision should contain a description of all changes made to the service in the new release

Summary

As your SOA initiative matures and more services are added to your portfolio, version management will become a larger and larger part of your governance concerns. In the early stages of your initiative consumers will need to know and understand how version management will work before they sign on to depend on services we part of their consuming applications.

This paper has described:

1. What needs to be versioned
2. Versioning requirements for a registry/repository
3. A versioning numbering scheme
4. Suggested governance policies for versioning