IBM Global Services

Message Specification

Version 2.3

Technique Paper

Document History

Document Location

Revision History

Date of this revision:	Date of next revision *(date)*

Revision Number	Revision Date	Summary of Changes	Authors
1.9	May 8th, 2008	First version	Abdul Allam, Amlan Sengupta
2.0	May 26th, 2008	Updates to section 3, 4 and addition of Section 7.0 – Estimation Guideline	Abdul Allam, Amlan Sengupta
2.1	July 27th 2008	Tie more to SOMA tasks/ activities/phases explicitly. High level : Identification:: Inform Modeling, Spec: Information Spec; Spec: Service Specification	Abdul Allam, Amlan Sengupta
2.2	August 12 2008	Added figure 3 – service consumer view of a service interface	Abdul Allam, Sengupta
2.3	August 16, 2008	Summasry SOMA steps figure added to Section 5.0. Comments from review incorporated	Abdul Allam

Table of Content

1. Introduction:

In SOA we construct business solutions using a building block approach across multiple architectural layers and functional domains. SOMA brings a methodical approach to the analysis and design of these building blocks, and especially focuses on the processes, the services and the service components. Each of these architectural elements encapsulates data and behavior behind a formalized interface so that business solutions can be assembled from a known set of reusable parts.

For this approach to be successful the elements must be carefully designed to ensure they provide clearly defined capabilities to provide for broad reuse of services across lines of buisness. The focus of this technique paper is Message specification which is a key activity in Service Specification. Message Specification, a key enabler of SOA, ensures that a well-constructed service contract is created

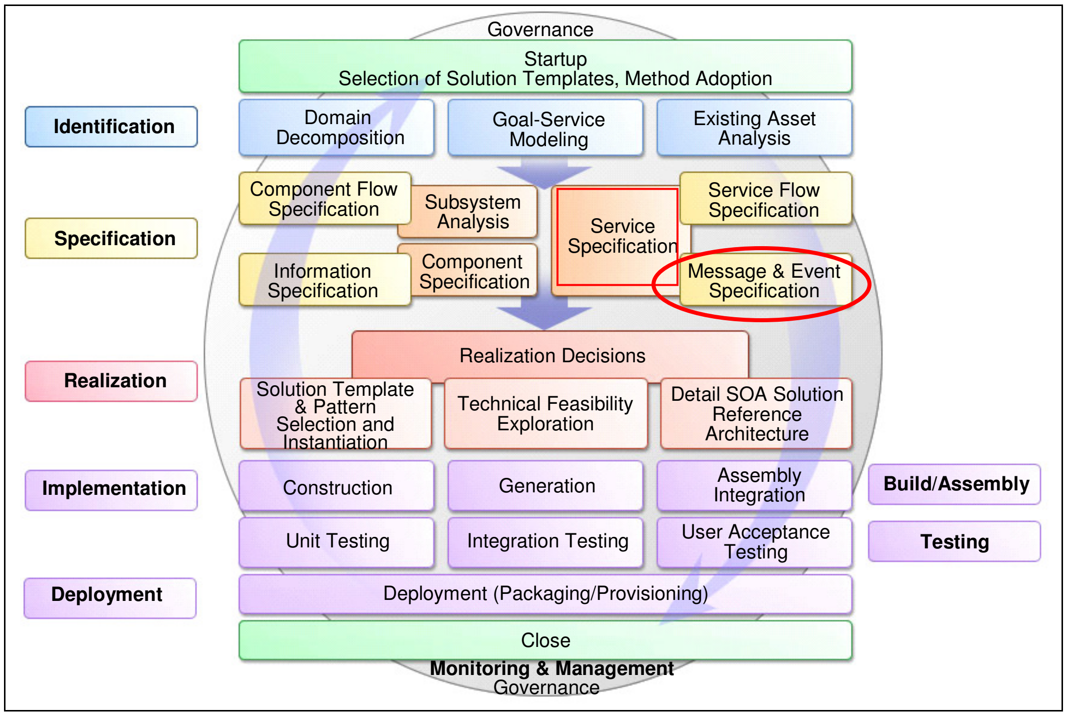


Figure 1: SOMA 3.1

2. Purpose

Purpose is to
1. describe the steps that need to be followed during service message specification
2. define used terms
3. provide relationship between various components in message specification

4. identify and recommend best practices
5. provide an end to end example to explain steps involved in the message specification

3. Description

Message specification is comprised of a set of tasks which help in creation of service contract and interface based on known standards and best practices. It is needed to define common message structures (request, response and error messages) to be used consistently across all service operations. Use cases and Information analysis serve as key input to message specification. Well-defined contracts are a core aspect of SOA and necessary for successful reuse, composability, minimized TCO (total cost of ownership). The more a service is reused, the more locked-in the contract becomes, necessitating appropriate attention to designing high-quality contracts. A consistent data model is also a key element of contracts, as it maximizes data reuse, reduces redundancy, and minimizes transformation overhead.

Figure 2, provides a high level view of the various components in a service interface.

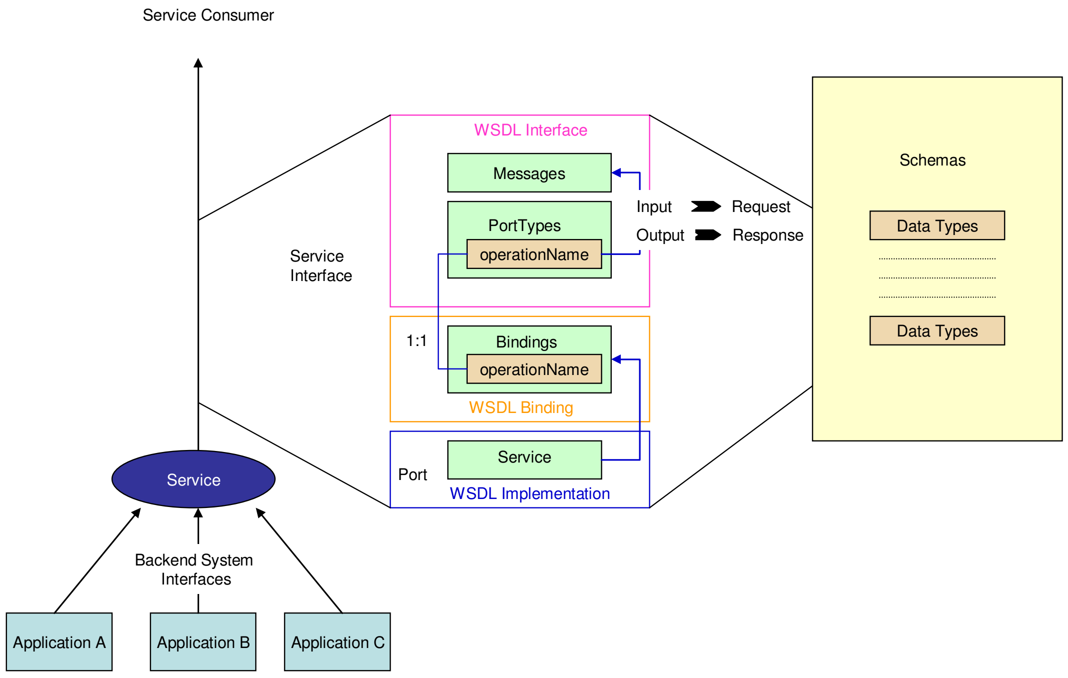


Figure 2: Message Specification Overview

The input/output messages of a service are represented as XML Schema. XML Schema specifies how services talk to one another. XML Schemas also specify a standard way of representing corporate data and a common message format for interactions with the service layer.
A XML schema is hierarchical in nature. Start with core data (enterprise wide applicability), business data (which has enterprise wide and cross domain utility),

operations data (for each service operation, getEmployeeInfo(Id, name) ; getProductInfo does not need employee id and name). Build the schema by starting from the core data and build up to the operation level data.

WSDL: ensures the wsdl, service interface is reusable and consistent across multiple consumers and thus meets their needs. WSDL specifies how they converse/interact. Schema define the data types exchanged between the service consumer ad provider.

Some of the key components in the service interface to focus on are:

- **Data**
- **Schema**
- **WSDL** also know as the Service contract

As we explore the details of a service interface, the following figure elaborates a Service consumer view.

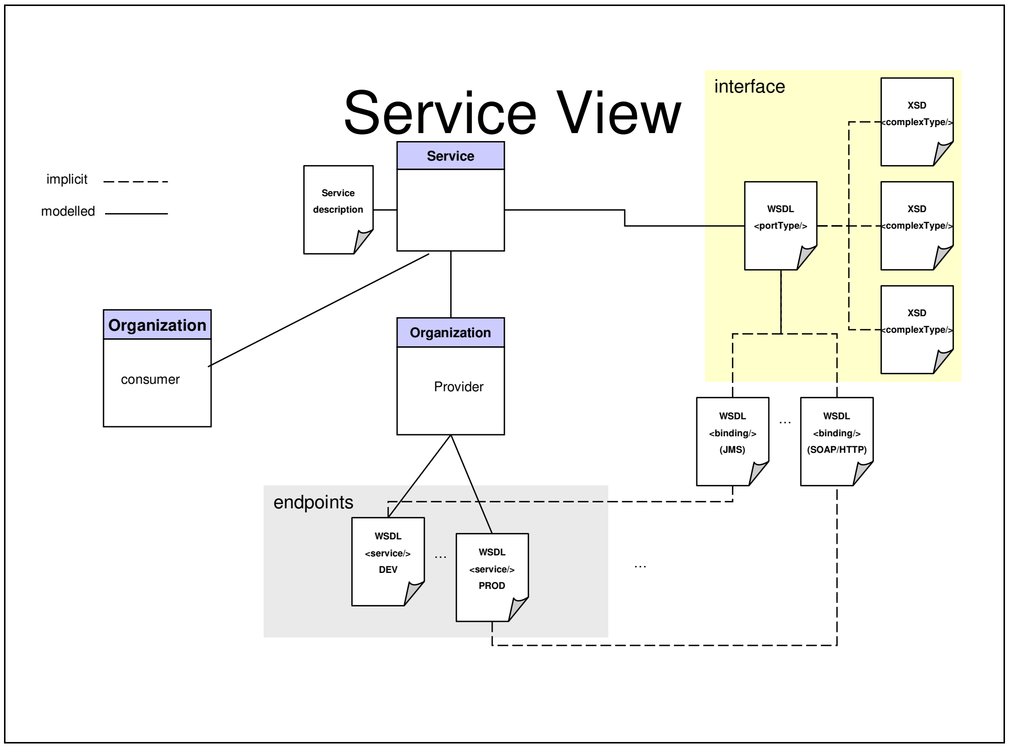


Figure 3: Service consumer view of a service interface

As details of each of these components are explored in this paper, the objective is to highlight the options to consider and rationalize why one is a better alternative than the other. The steps and examples elaborated in this document are based on best practices collected from client engagements. Before getting into the details on each of the above components, a recommended design approach is explored in the next section. These components will be described in the context of the recommended design approach.

3.1 Design Approaches

A design approach is a pre defined set of steps taken to achieve a delivery milestone. The use of a design approach is very beneficial as its helps in achieving consistency across solutions. It also helps in achieving quality solutions. Programming models and development tools based on XML and Web services use the following approaches for building Web services. We will examine the following three design approaches

- **Bottom up**
- **Top down**
- **Meet in the middle**

3.1.1 Bottom up Design (as part of Existing Asset Analysis)

All the integrated development environments (IDEs) provide tools for creating Web service implementations from existing code (for example, Java™ or COBOL). With this approach, the developer usually selects an existing JavaBeans or an EJB component and invokes a wizard that generates a WSDL file that can be used to invoke the bean or EJB as a Web service.

The bottom up development approach is appropriate when there is an existing body of legacy code (for example JavaBeans, EJB, COBOL, etc). This is done as part of Existing Asset Analysis of SOMA when existing assets including interfaces/interface specifications are available and analyzed. With this approach, you should carefully review interfaces of existing classes before generating the WSDL interface. If the Java interface contains any of the following, it can be considered weakly-typed:

- java.lang.Object used as either parameter or return type for a method – since java.lang.Object is not a supported JAX-RPC data type. Also it often gets mapped to <xsd:anyType> when used for parameters or return types.
- Collection classes (for example, java.util.Vector) used as either parameter or return type for a method (JAX-RPC constraint) – as they are not supported by JAX-RPC data types. It often gets mapped to literal array (generic array) when used for parameters or return types.

One should consider either re factoring the legacy code to ensure that interfaces are strongly typed or building a mediation module that will wrap the weakly typed interface with a strongly typed one.

Many skilled Java developers like to use bottom-up techniques to accelerate Web services development in SOA projects. They develop implementations for new services in Java first, and then use a wizard to create WSDL interfaces for these services. Although this approach can speed the implementation of individual services, it frequently means problems for the SOA project as a whole. Since in bottom-up approach we are using code driven contract generation, this may not reflect the business requirements which are a key aspect of SOA (business driven approach). As a result, often one ends up with brittle service contracts which are essentially not reusable. In addition, the bottom-up generation approach frequently results in schema construct definitions driven by Java patterns. This results in a contract with types which cannot be reused.

3.1.2 Top down Design – Business Requirements driven approach

In this approach, as shown in Figure 4 below there are various stages in the message specification development. The tasks on the Y-axis represents the SOMA tasks in Identification and Specification phases relevant to message specification.

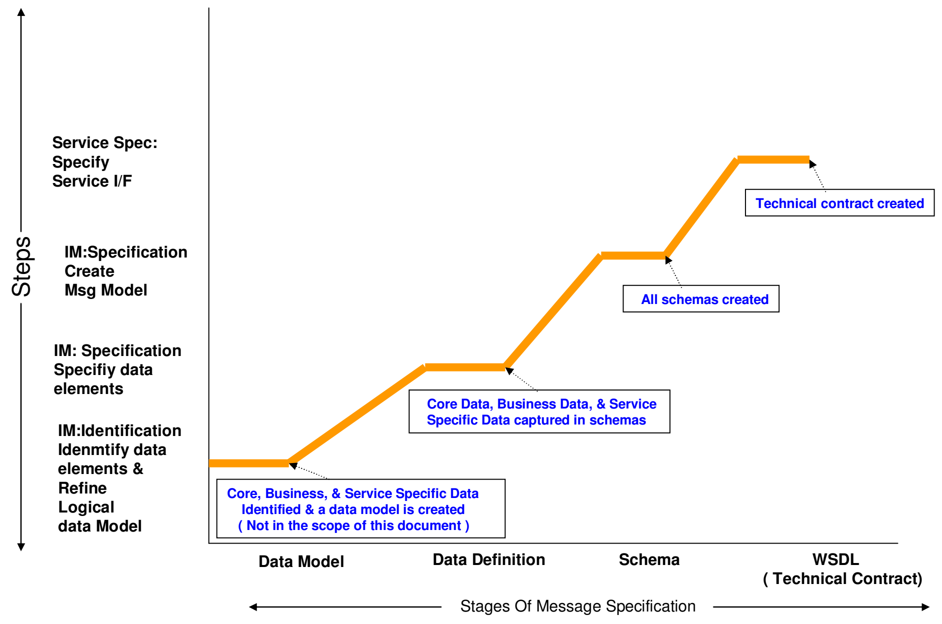
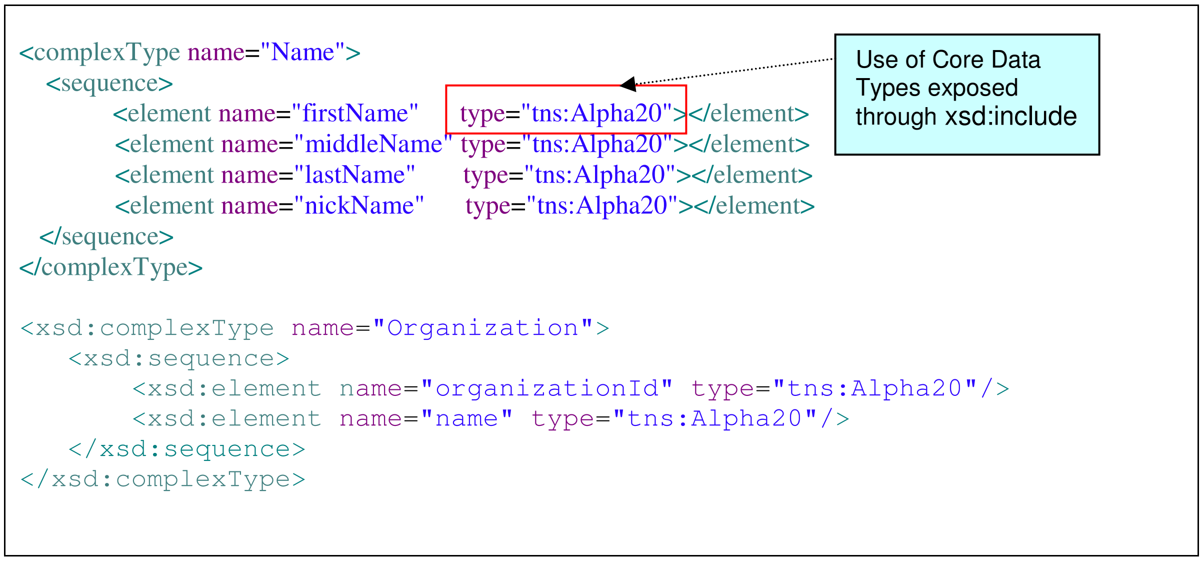

Steps
Service Spec: Specify Service I/F
IM:Specification Create Msg Model
IM: Specification Specifiy data elements
IM:Identification Idenmtify data elements & Refine Logical data Model
Technical contract created
All schemas created
Core Data, Business Data, & Service Specific Data captured in schemas
Core, Business, & Service Specific Data Identified & a data model is created (Not in the scope of this document)
Data Model
Data Definition
Schema
WSDL (Technical Contract)
Stages Of Message Specification

Figure 4: Stages of message development.

Data Model Stage: During the data model stage in the Message Specification process as shown in Figure 3 the architect and developer start by performing analysis on the data required for the service. The result of this analysis is a good understanding of various kinds of domain business objects e.g. Address, Employee, etc. that are required to support the business need. A logical data model would be a good source of information at this stage. At the end of this stage all the required business entities are identified.

Data Definition Stage: In the data definition stage, information and logical data models are explored to identify business entities and their relationships. Figure 5 below shows a business data type "Name" which has an element (example firstName).

Business data provides guidance on what core data should be. This is done by identifying commonality in the business data to arrive at core data types. The business data type "Name" uses a core data type "Alpha20" as the type of its elements (e.g. firstName). If there is any change to the properties of “Alpha20” in the future, all the elements in Figure 4 will get updated.



```
<complexType name="Name">
  <sequence>
        <element name="firstName"    type="tns:Alpha20"></element>
        <element name="middleName"  type="tns:Alpha20"></element>
        <element name="lastName"     type="tns:Alpha20"></element>
        <element name="nickName"     type="tns:Alpha20"></element>
  </sequence>
</complexType>

<xsd:complexType name="Organization">
   <xsd:sequence>
       <xsd:element name="organizationId" type="tns:Alpha20"/>
       <xsd:element name="name" type="tns:Alpha20"/>
   </xsd:sequence>
</xsd:complexType>
```

Use of Core Data Types exposed through xsd:include

Figure 5: Business Data Types are built by including Core Data Types

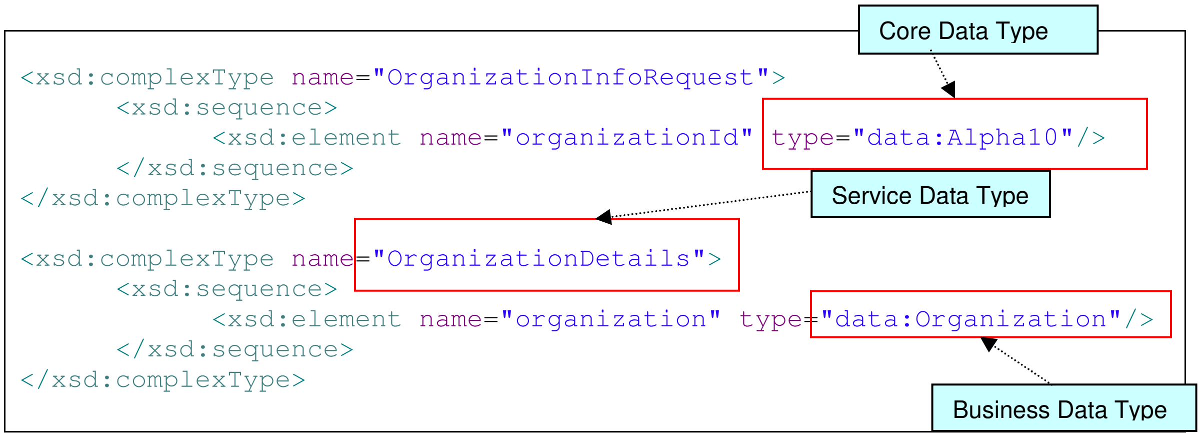
The developer would now verify whether the required core data types are available at enterprise level. If they are not available, it will need to be captured as required core data. An example of a code data type “Alpha20” is shown in Figure 6. These core data types would be used by the service contract.

```
<xsd:simpleType name="Alpha20">
        <xsd:restriction base="xsd:string">
        <xsd:maxLength value="20"/>
        </xsd:restriction>
</xsd:simpleType>
```

Figure 6: Example of a Core Data Type

The identified business entities are now mapped to the service operations (discussed later in Figure 6). The objective of this mapping is to identify business data which are required to support the service operations.

Now that core data type and business data type have been identified, we need to consider service specific data. As a result the identified business entities are now mapped to the service operations (discussed in Figure 7). The objective of this mapping is to identify business data which are required to support the service operations. Figure 7 also shows examples of service specific data- “OrganizationInfoRequest and OrganizationDetails”. The primary source for this information is business process and uses cases.



```
<xsd:complexType name="OrganizationInfoRequest">
      <xsd:sequence>
            <xsd:element name="organizationId" type="data:Alpha10"/>
      </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="OrganizationDetails">
      <xsd:sequence>
            <xsd:element name="organization" type="data:Organization"/>
      </xsd:sequence>
</xsd:complexType>
```

Figure 7: Service Specific Data Type

The relationships between all the data types discussed so far is shown in Figure 8.

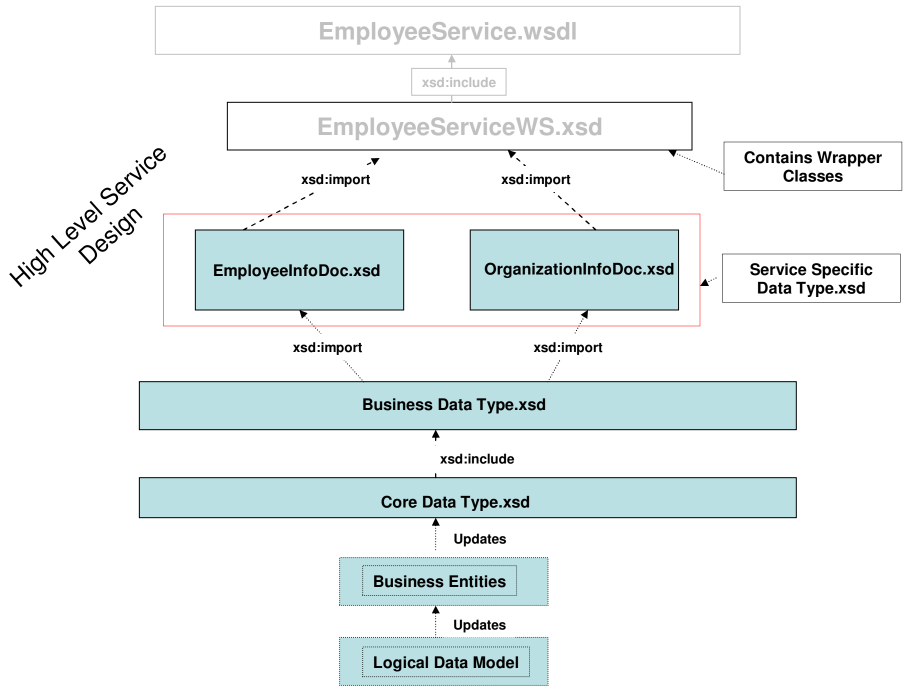


Figure 8: Data Types

Schema Stage: As part of the next stage, all the schemas required for the service are captured in the form of XSDs. It is important that good schema design practices are enforced during this stage to ensure development of re-usable interfaces. As part of this best practice xsd:include is used when the complex and the simple types in the included schema is exposed under the parent name space. Import is used as a practice when the child schema needs to remain in its own namespace. The parent / the importing schema uses a reference to the child schema, before referring to the members of the importing schema.

WSDL Stage: With all the data types captured and defined in the schemas, we are now ready to generate the WSDL or a service contract. For this process a developer can use an IDE to create the WSDL.

Key advantages of this Top Down approach are:

- You start by looking at data in the context of business requirements.
- You design contract from scratch. This helps in creation of services with bindings which don't break at run time.
- You design the operations you need, and the input and output parameters that are expected or required.
- Your data becomes the building block for the schema development.
- Ease of integration with business partners who have adopted the same approach. If well know formats are used, integration with other business domains become seamless.
- Reusing such standards can help reduce the amount of work required in developing semantically transparent data formats.

3.1.3 Meet in the Middle Design Approach

This approach involves a combination of the previous two. The developer first defines the service interface using WSDL and XSD, and generates a skeletal implementation for the service. If necessary, the developer may also use a bottom-up technique to expose existing code using a convenient application programming interface (API), COBOL Copybook or EJB. The developer may write code (typically wrapper or adapter) or use existing adapters supported in a data appliance (e.g. DataPower), that performs transformation between the newly designed interface and the old interface. This approach is further elaborated in the following figure.

Figure 9 summarizes the above discussion on various design approaches and brings out some of the key differences between them. This is then followed up with a best approach recommendation.

Top Down	Bottom Up	Meet in the middle
In this approach business requirements drive the contract design. The design is started from scratch based on business requirements. ***This is a good practice.***	This approach generates your contract from JAVA code or existing legacy code. Essentially developing a contract which is not driven by business	In this approach, you would ideally write the contract from scratch and align its implementation with the existing application code. The advantage here is we

	requirements. This could lead to service contracts which may not be re-usable. ***This is not a good practice.***	are leveraging existing code base. By leveraging existing code we may risk inheriting existing bad practices for e.g. support data elements that are not required. In addition we would require wrappers or adapters to be built to bridge the new and existing interfaces.
Business analyst models business processes and Business data is captured as Business Entities which are used to drive service contract. ***This is a good practice***	Cobol copy book or Database tables are often used to derive a service contract. ***This is a bad practice since it is constrained by existing data model.***	XSD which is exported from business entities is used to create the service contract. It is a good practice. However, you may still need to create wrappers or adapters.

Figure 9 Design approach comparisons

Recommended Best Practice: **Use the top down approach, to design service interface using XSD and WSDL, and from this generate skeletal Java code. This approach is based in business requirements and hence the service interface will be re-usable across projects/domains. Note, if COBOL Copy book exists then we need to use XSD and WSDL created from the top down analysis to enhance the copy book to support the service interface.**

3.2 Message Components

In this section, we will discuss some of the key components of a Message.

3.2.1 Data

Data is very important part of an Enterprise and its solutions. Hence, it is very essential to examine carefully the data or sets of data which are needed to support Service specification components such as Schema and WSDL. This section will focus on key aspects of data such as Data types, Data encapsulation, Data definitions, etc. and associated best practices.

3.2.1.1 Data Types

One of the main advantages of schema definitions is it reusability. In an attempt to reuse schema definitions, data types are created and these are broken down into following categories.

- **Core Data Types**
  Core Data Types are data items that describe common concepts which can be re-used across various business entities. These core data types are reused at the Business Data Type level through 'xsd: include' in the core data type schema. The following is an example of reusable core data type through use of include. It Core Data Type definitions serve as a container for properties.

```
<xsd:simpleType name="Alpha20">
            <xsd:restriction base="xsd:string">
            <xsd:maxLength value="20"/>
            </xsd:restriction>
</xsd:simpleType>
```

Figure 10: Core Data Type

- **Business Data Types**

  Business Data Types are specific data items that define common concepts used in a business domain, e.g., Human Resources. They are mainly used to capture and facilitate the reuse of domain information. Business Data Types also reuses Core Data Types as shown in the Figure 11.

```
<complexType name="Name">
   <sequence>
     <element name="firstName"  type="tns:Alpha20"></element>
          <element name="middleName" type="tns:Alpha20"></element>
          <element name="lastName"   type="tns:Alpha20"></element>
          <element name="nickName"   type="tns:Alpha20"></element>
  </sequence>
</complexType>
```

Use of Business Data Types which also reuses Core Data Types exposed through xsd:include

Figure 11: Business Data Type

- **Service Specific Data Types**

  Service Specific Data Types are data items that define the data type's specific to the operations defined in the service contract. These data include schema constructs which sets up the request and the response message structure for the respective service operations. Service Specific Data Types also reuses Business Data Types as shown in Figure 12.

```
<! -- These elements enable the document/literal/wrapped convention. -->
<! -- These types define the structure of request and response      -->

<xsd:element name="getEmployeeInfoRequest" type="tns:EmployeeInfoRequest" />
<xsd:complexType name="EmployeeInfoRequest">
    <xsd:sequence>
                  <xsd:element name="employeeId" type="data:Alpha10"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="getEmployeeInfoResponse" type="tns:EmployeeResponse" />
<xsd:complexType name="EmployeeResponse">
    <xsd:sequence>
        <xsd:element name="employee" type="data:Employee"/>
        <xsd:element name="organization" type="data:Organization"/>
    </xsd:sequence>
</xsd:complexType>
```

Figure 12: Service Specific Data Type

Now that we have talked about data, the following section explores aspects of data encapsulation at the XML Schema level and the best practices we can use.

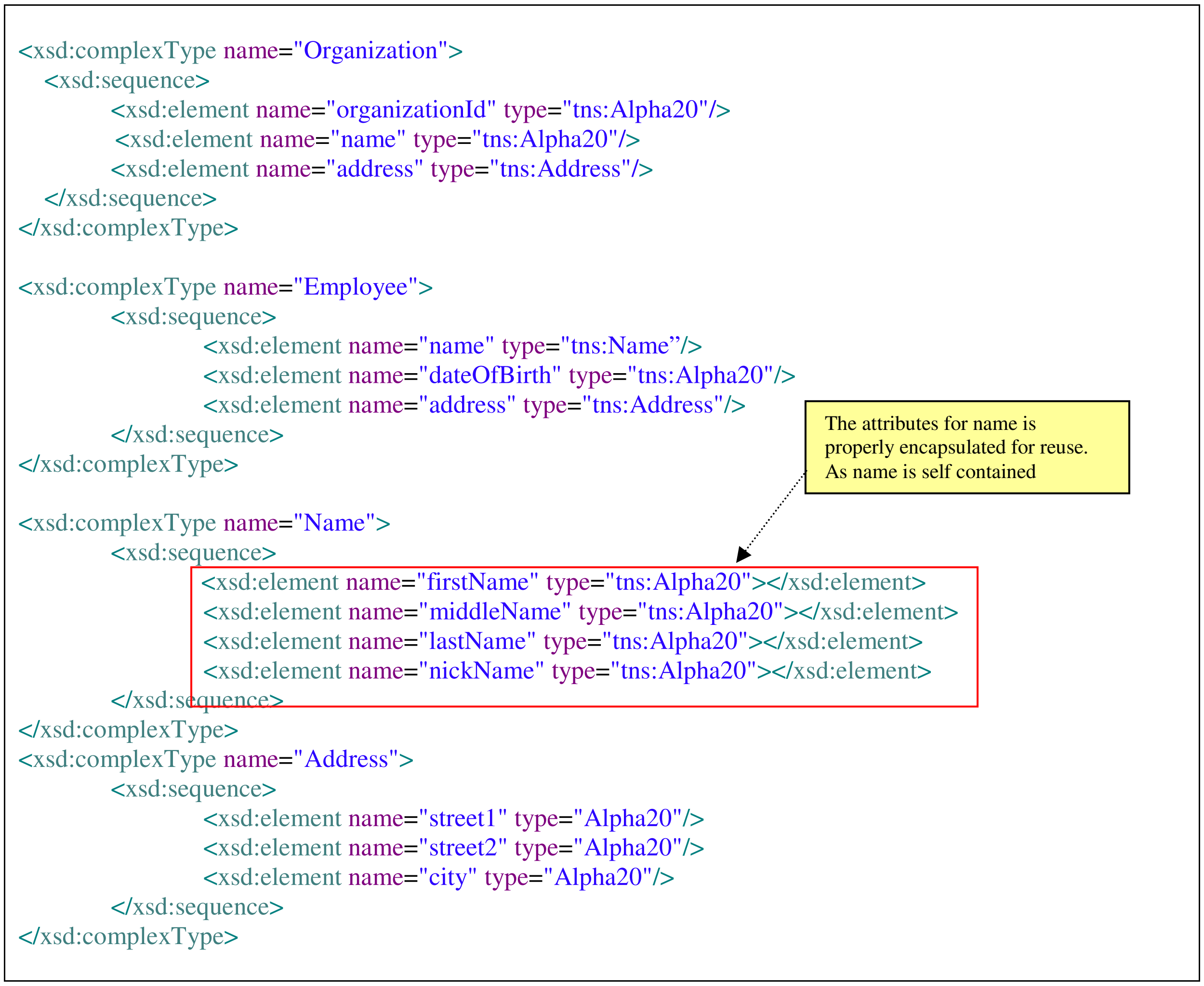
3.2.1.2 Data Encapsulation

Data encapsulation, sometimes referred to as data hiding, is the mechanism whereby the implementation details of a data are kept hidden. Let's start by examining the following example in Figure 13. In this example, we have business entity "Employee" composed of name and address related information. The name and address elements are not re-usable if they are needed in other business entities. This limitation in re-usability is due to the elements being encapsulated within the complex type "Employee". By encapsulating we have access only to Complex type "Employee" and not the sub entities - name and address.

```
<xsd:complexType name="Employee">
        <xsd:sequence>
                <xsd:element name="firstName" type="Alpha20"/>
                <xsd:element name="lastName" type="Alpha20"/>
                <xsd:element name="dateOfBirth" type="Alpha20"/>
                <xsd:element name="street1" type="Alpha20"/>
                <xsd:element name="street2" type="Alpha20"/>
                <xsd:element name="city" type="Alpha20"/>
        </xsd:sequence>
</xsd:complexType>
```

Figure 13: Schema constructs are valid with poor reusability.

Let us now examine Figure 14. Here we have two new sub entities called "Name" and "Address" which are both used by "Employee" and "Organization". Thus compared to scenario in Figure 10, we are achieving good reusability for sub entities “Name” and “Address”. They can now be used by any entity.



```
<xsd:complexType name="Organization">
   <xsd:sequence>
          <xsd:element name="organizationId" type="tns:Alpha20"/>
           <xsd:element name="name" type="tns:Alpha20"/>
          <xsd:element name="address" type="tns:Address"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Employee">
          <xsd:sequence>
                    <xsd:element name="name" type="tns:Name”/>
                    <xsd:element name="dateOfBirth" type="tns:Alpha20"/>
                    <xsd:element name="address" type="tns:Address"/>
          </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Name">
          <xsd:sequence>
                    <xsd:element name="firstName" type="tns:Alpha20"></xsd:element>
                    <xsd:element name="middleName" type="tns:Alpha20"></xsd:element>
                    <xsd:element name="lastName" type="tns:Alpha20"></xsd:element>
                    <xsd:element name="nickName" type="tns:Alpha20"></xsd:element>
          </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Address">
          <xsd:sequence>
                    <xsd:element name="street1" type="Alpha20"/>
                    <xsd:element name="street2" type="Alpha20"/>
                    <xsd:element name="city" type="Alpha20"/>
          </xsd:sequence>
</xsd:complexType>
```

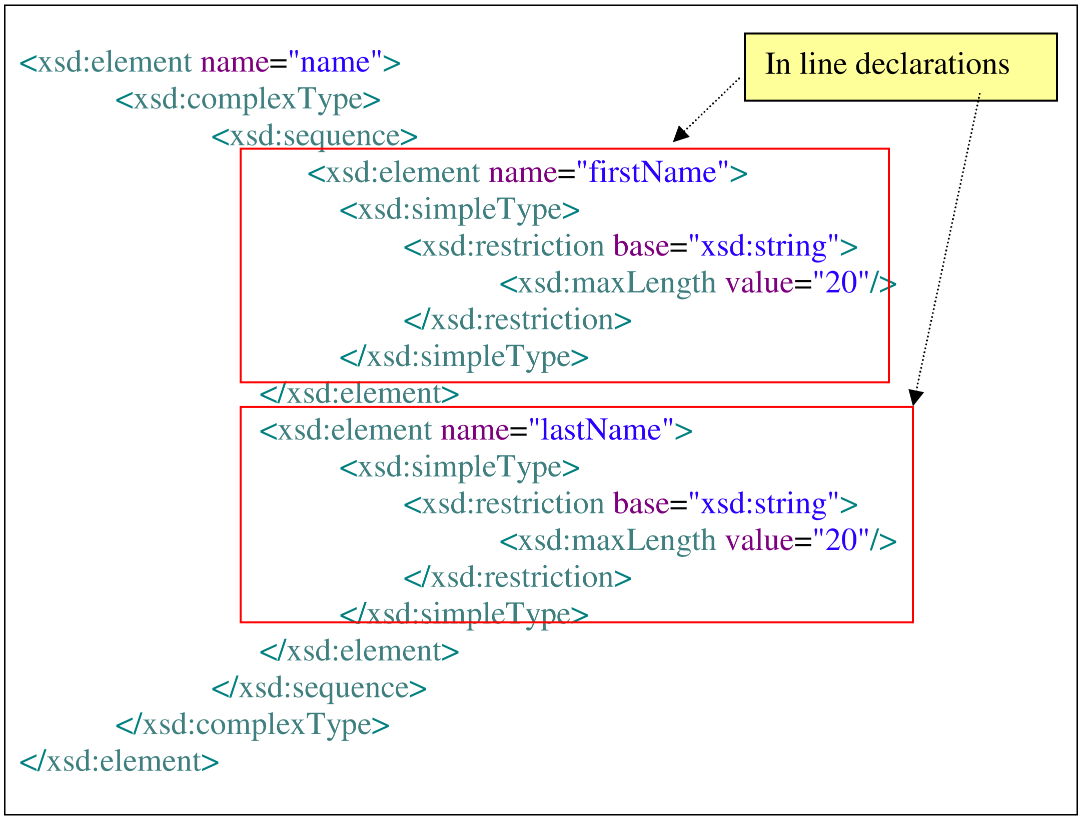
The attributes for name is properly encapsulated for reuse. As name is self contained

Figure 14: Schema constructs are valid with good reusability - Strongly recommended.

In the next section we discuss some of the best practices associated with data declaration.

3.2.1.3 Defining Elements

XML Schema supports a number of methods for defining data. In the example b, the definitions for all the elements are defined inline. As the name implies, inline, or local, definitions are included directly in the element declaration. In Figure 12 due to declaration of “firstName” and “lastName” in the scope of the element “name”, these complex types are not accessible. If there is other business entity or sub entity that may have a use for “firstName” and “lastName” will not be able to reuse them. In addition if there is a need for example to apply or change restrictions on “20“character fields, it would be necessary to apply these changes in every 20 character reference.



```
<xsd:element name="name">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element name="firstName">
                           <xsd:simpleType>
                                 <xsd:restriction base="xsd:string">
                                         <xsd:maxLength value="20"/>
                                 </xsd:restriction>
                           </xsd:simpleType>
                    </xsd:element>
                    <xsd:element name="lastName">
                           <xsd:simpleType>
                                 <xsd:restriction base="xsd:string">
                                         <xsd:maxLength value="20"/>
                                 </xsd:restriction>
                           </xsd:simpleType>
                    </xsd:element>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Figure 15: Inline Element Definition Snippet – Poor Practice

In the example shown in Figure 15, both 'xsd:simpleType' and 'xsd:complexType' are used**.** Even with white space added for readability, the resulting schema is shorter than that of the inline variety with an overall structure of Name, which is much easier to understand. For instance, any changes made to the core data type Alpha20 gets reflected in all its references in Figure 15. Also "firstName" and "lastName" now becomes a reusable sub entity which can be referred by others and has a potential for further reuse. Hence it is strongly recommended that declarations of inline elements be avoided.

```
<xsd:element name="name" type="tns:Name"/>
<xsd:complexType name="Name">
        <xsd:sequence>
                <xsd:element name="firstName" type="tns:Alpha20"/>
                <xsd:element name="lastName" type=" tns:Alpha20"/>
        </xsd:sequence>
</xsd:complexType>
```

Figure 16: Inline Element Definition Snippet – Good Practice

Now that we have discussed the best practices about declaring elements, we will next focus on best practices associated with Schemas.

3.2.2 Schema

Industry groups have started to work on XML-based standards for various types of information. This effort involves designing of core data types which used by business data types through “xsd:include”. These business data types in turn get used by the operations through “xsd:import” as shown in Figure 20. This process results in creation of a Schema for a specific service operation. This process is explained in more detail in the following section.

3.2.2.1 Operation driven schema design

In top down schema design the business requirements and business processes drive the identification of business entities. These business entities are used in the creation of service contracts. The service contracts created incorporate schemas of core, business and service specific data types. The service contracts consist of multiple components. One of the key component is service operation. The types of operation that can be used in a service contract or WSDL are shown in the following table.

Operation Type	Description	Order of Messages
request-response	client calls Web service – response expected	Input output fault
solicit-response	Web service solicits client – response expected	output input fault
one-way	client calls Web service – no response expected	Input
Notification	Web service calls client – no response expected	Output

Figure 17: Service Contract or WSDL Operations.

In order to support various operation types, it is recommended that, schemas be modularized based on respective operations. In the example below, we have a WSDL “EmployeService.wsdl” for the service “EmployeeService”. This service has two operations – “getEmployeeInfo” and “getOrganizationInfo”. The following diagram elaborates the operation “getEmployeeInfo” and the schema constructs (input/output messages) that supports it.

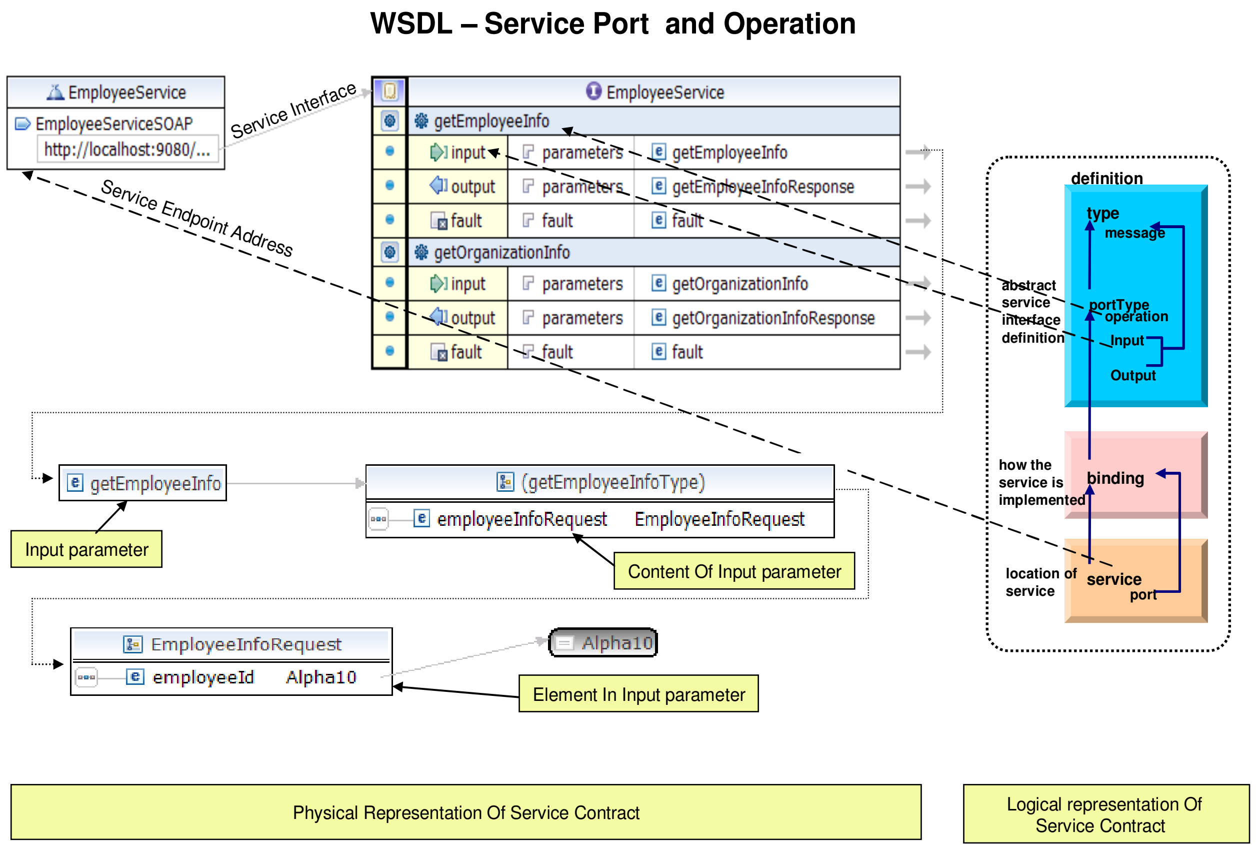


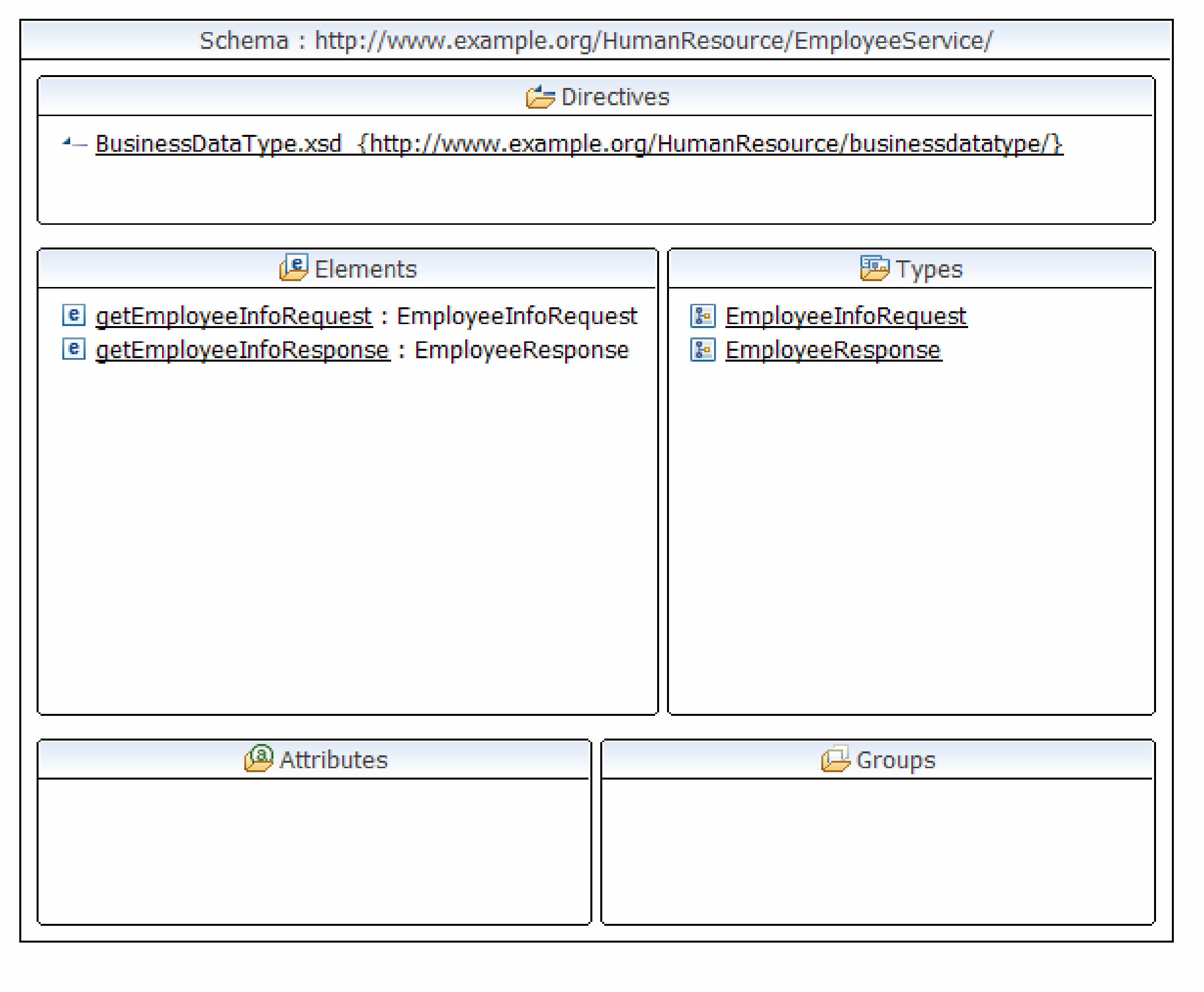
Figure 18: WSDL Port and Operations.

Note: In the above figure service end point details are aspects of SOMA Realization and Implementation. We have shown here the service point realization for illustration purposes. However, in the event of leveraging an existing service, we need to ensure that this level of detail is available.

In Figure 18, the two operations associated with this service contract are getEmployeeInfo and getOrganizationInfo. For the operation "getEmployeeInfo" in Figure 18, we recommend the creation of EmployeeInfoDoc.xsd whose contents are elaborated below and in Figure 19. The following Figure 19 has three main sections which are

- Directives – which has all includes and imports done in this schema. In this scenario BusinessDataType.xsd has been imported into the parent schema.
- Elements – declare references to the request and the response document for the operation getEmployeeInfo
- Types – declare the data types for the request and the response documents which the elements refer to

The key advantage of this "operation driven schema design" is, any changes that happen to this operation during the service life cycle can be contained within the schema. In other words, any changes to the service specific data or its types have no impact on other operations associated with this service.


Schema : http://www.example.org/HumanResource/EmployeeService/

Directives

BusinessDataType.xsd {http://www.example.org/HumanResource/businessdatatype/}

Elements

getEmployeeInfoRequest : EmployeeInfoRequest

getEmployeeInfoResponse : EmployeeResponse

Types

EmployeeInfoRequest

EmployeeResponse

Attributes

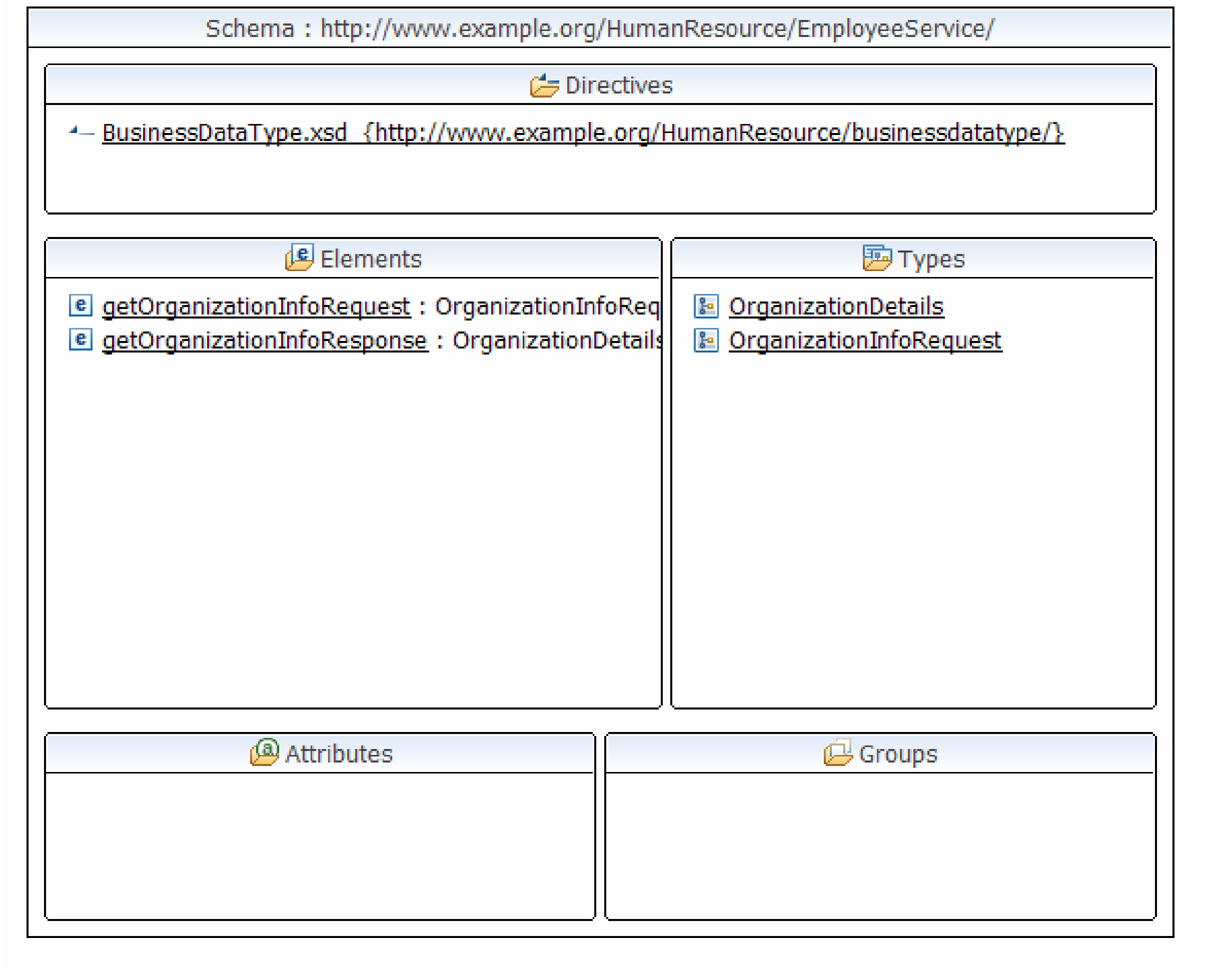
Groups

Figure 19: XSD supporting operation

The second operation in Figure 20 is "getOrganizationInfo". To support this second operation we create "OrganizationInfoDoc.xsd" whose contents are shown below. The following Figure 20 has three main sections which are

- Directives – which has all the includes and imports done in this schema. In this scenario BusinessDataType.xsd has been imported into the parent schema.
- Elements – declare references to the request and the response document for the operation getOrganizationInfo
- Types – declare the data types for the request and the response documents which the elements refer to

The advantage of "operation driven schema design" applies in this case as well. In other words, any changes to the service specific data or its types have no impact on other operations associated with this service.


Schema : http://www.example.org/HumanResource/EmployeeService/
Directives
Elements
Types
Attributes
Groups

BusinessDataType.xsd {http://www.example.org/HumanResource/businessdatatype/}

getOrganizationInfoRequest : OrganizationInfoReq
getOrganizationInfoResponse : OrganizationDetails

OrganizationDetails
OrganizationInfoRequest

Figure 20: XSD supporting operation.

Note: Figures 19 and 20 deal with realization and implementation aspects, we have provided these as example for illustration purpose.

Now that we have discussed the service, its operations and the associated schemas, we next need to show the relationship between the schemas, the operations and the service contracts. This relationship is shown in Figure 21.

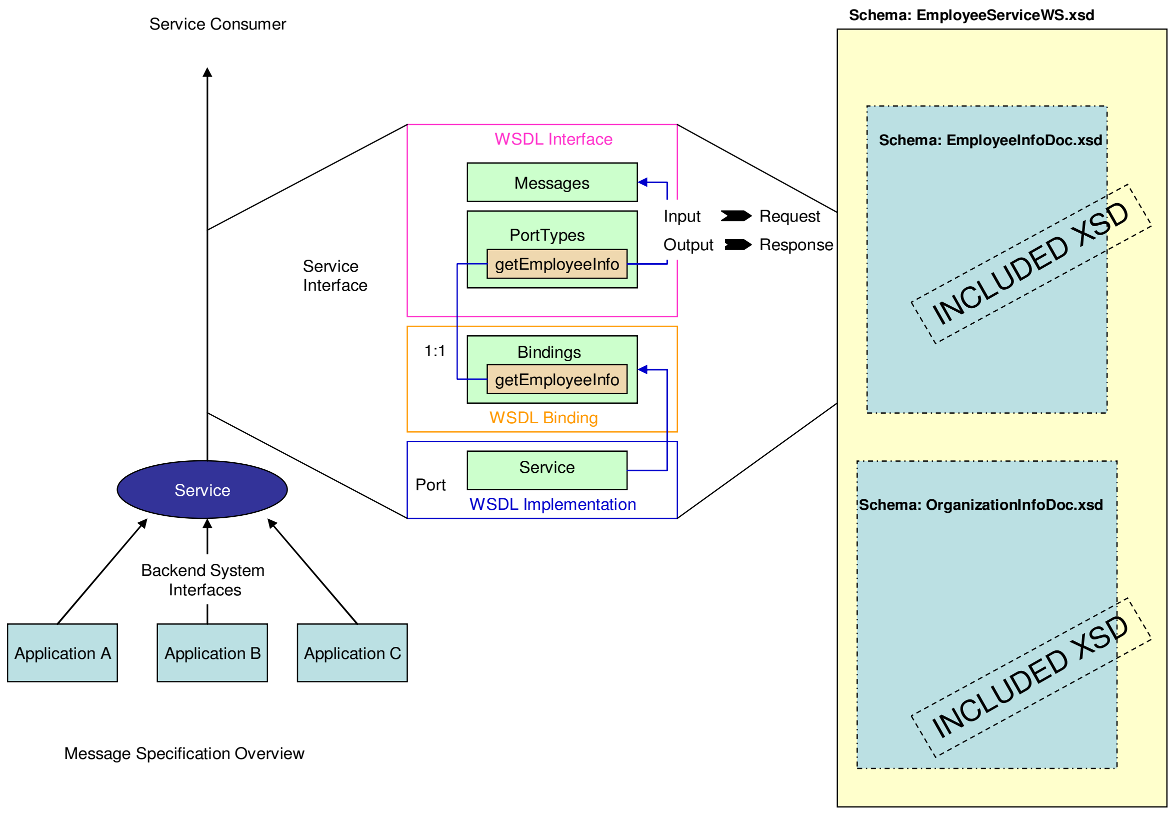


Figure 21: XSD imports to support service interface.

As we explore best practices in schema design, we need to be aware of advantages and disadvantages of abstract implementations. In the following section we discuss the aspects of abstract implementation.

3.2.2.3 Abstract Implementations

Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only information which is relevant for a particular purpose. While implementing Service Oriented Architecture, as best practice every attempt should be made to avoid exposing implementation details in an XML document.

In the following Figure 22, we are using an enumeration of error, warning, info and debug which are used to capture various states of logging by the application. As a result the error, warning, info and debug are tightly coupled to the consumer implementations. So any change in the future would require changes to the existing schema. These changes to the schema would need regeneration of service artifacts. This also can considerably increase the risks breaking of backward and binary compatibility at runtime. As a best practice, the error, warning, info and debug should be mapped at the application level to relevant logging states. This is a very good practice as the schema in Figure 21 is abstracted from the implementation specific details of the service. In future, any changes to mapping of error, warning, info and debug at the application level can be done without changes to the schemas.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
            targetNamespace="http://www.example.org/ws/HumanResource/EmployeeService/"
            xmlns:tns="http://www.example.org/ws/HumanResource/EmployeeService/"
            elementFormDefault="qualified" attributeFormDefault="unqualified">

  <xsd:complexType name="Fault">
    <xsd:sequence>
      <xsd:element name="faultDetails" type="tns:ErrorCode" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="ErrorCode">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="error"/>
      <xsd:enumeration value="warning"/>
      <xsd:enumeration value="info"/>
      <xsd:enumeration value="debug"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Figure 22: Abstraction

3.2.3.3 Use of Faults and Exception Handling

Source *Russell Butek* (*butek@us.ibm.com*), Developer, IBM

The term "exception" is technology neutral (though some technologies, like Java, also use the term). An exception is a response which indicates an exception condition – i.e, non-normal – response.

An interface is technology neutral, so the term "exception" appears in an interface's namespace. In the Web services world, exceptions are called faults, so in a service's Web service namespace we use the term "fault". There are a set of common elements and types defined in the file Exception.xsd, shown in listing 1.

Listing 1: Exception.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:tns="http://www.example.org/2008/09/exception/"
```

```
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                targetNamespace= "http:// www.example.org/2008/09/exception/"
                elementFormDefault="qualified"
               attributeFormDefault="unqualified">

          <xsd:element name="fault" type="tns:Exception"/>
          <xsd:element name="svcFault" type="tns:SvcException"/>

          <xsd:complexType name="Exception">
                    <xsd:sequence>
                              <xsd:element name="reasonCode" type="tns:Alpha8"/>
                              <xsd:element name="reasonText" type="tns:Alpha70"/>
                    </xsd:sequence>
          </xsd:complexType>

          <xsd:complexType name="SvcException">
                    <xsd:complexContent>
                              <xsd:extension base="tns:Exception"/>
                    </xsd:complexContent>
          </xsd:complexType>

          <xsd:simpleType name="Alpha8">
                    <xsd:restriction base="xsd:string">
                              <xsd:maxLength value="8"/>
                    </xsd:restriction>
          </xsd:simpleType>

          <xsd:simpleType name="Alpha70">
                    <xsd:restriction base="xsd:string">
                              <xsd:maxLength value="70"/>
                    </xsd:restriction>
          </xsd:simpleType>
</xsd:schema>
```

Exception is the parent for all exceptions. SvcException, which extends Exception, is the parent for all service-specific exceptions. These two exceptions can also be thrown by the infrastructure.

With this background, we have the following rules.

Exception naming rules

For an exception named Yyy in the Service Requirement Document:

- In the xxx.wsdl file:
  - The message for Yyy is named YyyFaultMsg;
  - The fault name in the operation is named YyyFault.
- In the xxxWS.xsd file:
  - The element which the WSDL message refers to is called yyyFault.
- In the xxxDoc.xsd file:
  - The type which the xxxUS.xsd's yyyFault refers to is named YyyException.

Exception hierarchy rules

- Each service must have a base service exception.
- This base service exception must extend SvcException.
- Any and all service-specific exceptions must extend this base exception.

  (Note: the next bullet discusses the WSDL, so instead of the term "exception" we will now use the term "fault", where each fault refers, according to the exception naming rules, to its corresponding exception.)
- Each WSDL operation must contain the following faults: fault, svcFault, the base service fault. Each WSDL operation may also contain some set of service-specific faults.

3.2.3 WSDL

The WSDL provides a standardized, complete service contract that potential customers can use to access the service, without the knowledge of how the actual service logic is implemented. Figure 23 shows the key constituents of a service interface or service description.

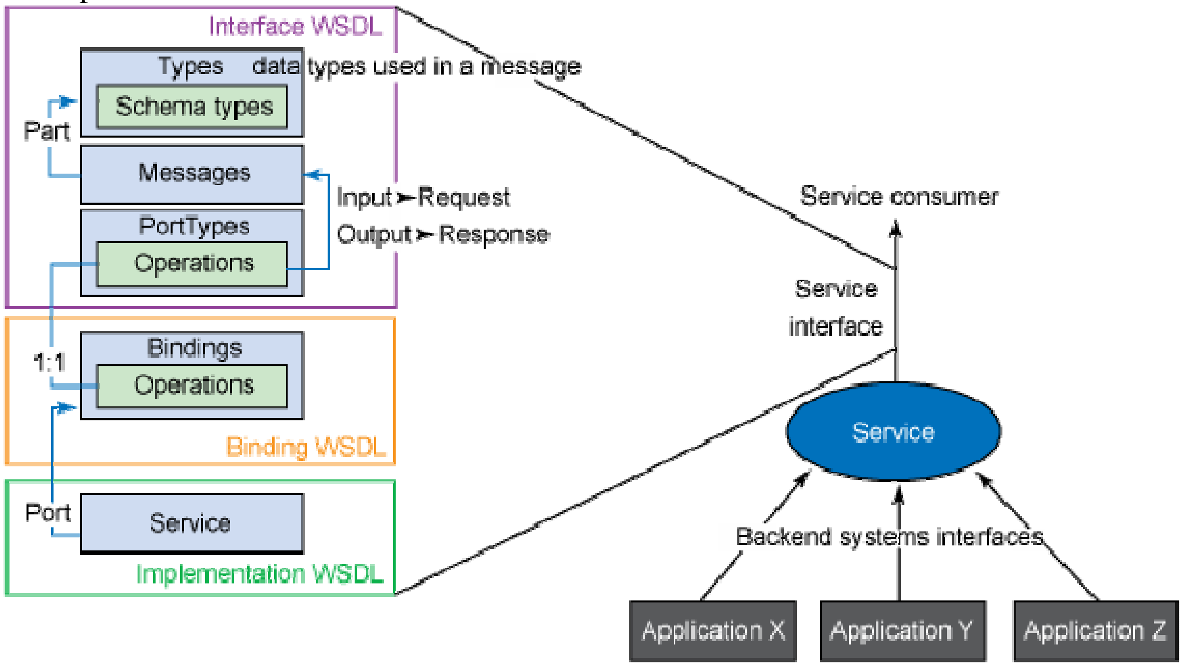
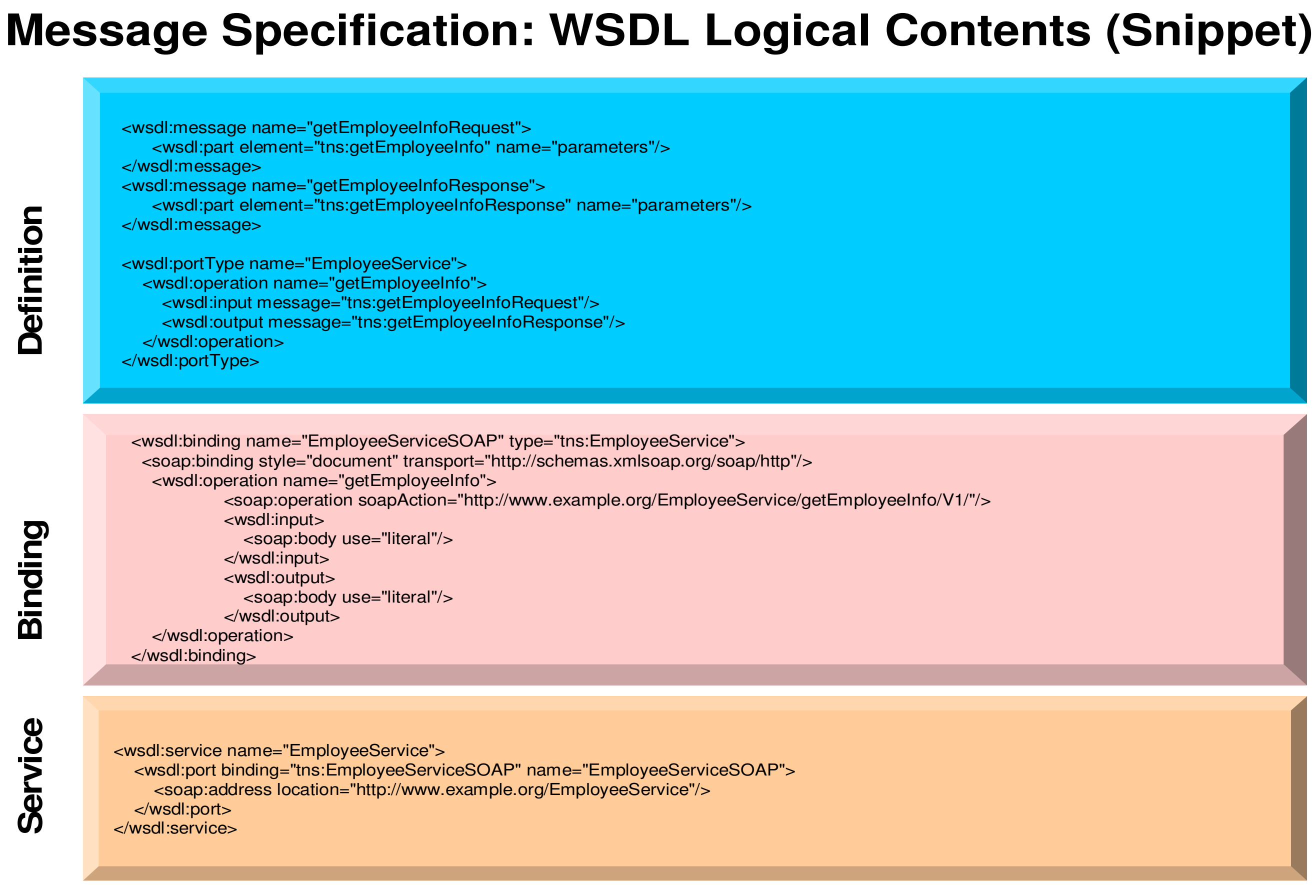


Figure 23: Service Component Model

Identifying and then specifying services is a core part of establishing a Service-Oriented Architecture. It starts with assessing and decomposing key business processes. This decomposition leads to identification of a set of services, each represented by a WSDL definition. The following Figure 24 shows a WSDL with some of its primary components.


Message Specification: WSDL Logical Contents (Snippet)
Definition
Binding
Service

```
<wsdl:message name="getEmployeeInfoRequest">
     <wsdl:part element="tns:getEmployeeInfo" name="parameters"/>
</wsdl:message>
<wsdl:message name="getEmployeeInfoResponse">
     <wsdl:part element="tns:getEmployeeInfoResponse" name="parameters"/>
</wsdl:message>

<wsdl:portType name="EmployeeService">
   <wsdl:operation name="getEmployeeInfo">
      <wsdl:input message="tns:getEmployeeInfoRequest"/>
      <wsdl:output message="tns:getEmployeeInfoResponse"/>
   </wsdl:operation>
</wsdl:portType>
```

```
<wsdl:binding name="EmployeeServiceSOAP" type="tns:EmployeeService">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getEmployeeInfo">
              <soap:operation soapAction="http://www.example.org/EmployeeService/getEmployeeInfo/V1/"/>
              <wsdl:input>
                 <soap:body use="literal"/>
              </wsdl:input>
              <wsdl:output>
                 <soap:body use="literal"/>
              </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

```
<wsdl:service name="EmployeeService">
   <wsdl:port binding="tns:EmployeeServiceSOAP" name="EmployeeServiceSOAP">
      <soap:address location="http://www.example.org/EmployeeService"/>
   </wsdl:port>
</wsdl:service>
```

Figure 24: WSDL Overview

So far we discussed the key components of a service and the best practices, in the next section we will explore how these best practices are applied during interface design and what are the benefits realized.

3.2.3.1 Good interface design

A good interface design constitutes the following:

- **Readability**: It is important that, the XML documents and the SOAP messages; are readable and decipherable as other designers/developers will be reading the WSDL, and the schema that defines the interface.

- **Consistency**: Consistency should be maintained in casing, naming, structure, and use of attributes to make service contracts and schema constructs legible. This is probably the simplest way to improve interface and service contract.
- **Support of validation:** Having an interface than can be validated is critical.
- **Abstraction of implementation details**: Interfaces should be designed based on business requirements with little or no reference to implementation details.
- **Logical organization of data**: A well-organized set of data help in creating proper messages.

3.2.3.2 Use of Document/literal wrapped convention

Document/literal/wrapped (DLW or doc/lit/wrapped) is a convention defined by Microsoft, and supported by default by .NET, that 'wraps' a document/literal binding in a single element whose name matches the Web Service operation. Hence for a service to be WSI (Web Services Interoperability guide lines) compliant, only Document/literal/wrapped should be for development purposes as other use and styles are not supported by .Net. Document/literal wrapped convention has the following attributes:

- The message has a single part that identifies an element as the content in the message. This element is the wrapper which points to the content of the request and the response messages.
- This element has the same name as the name of the operation. This helps the SOAP engine on the service provider side to use the method name to determine how to structure the response to the original request.
- Document literal is WS-I compliant, and the wrapped pattern meets the WS-I restriction that the SOAP message's soap:body have only one child.

Key advantages of this convention are:

- Everything that appears in soap:body is defined in the schema. Thus this schema can be used to validate the message.
- The method name appears in the SOAP message. This is a big advantage as the SOAP engine in which the service provider is running will be able to use the method name to determine how to structure the response to the original request. .

Key disadvantages of this convention are:

- To support document literal wrapped convention, the schemas or the WSDL supporting the schemas should declare all the wrappers that are needed to construct a document literal wrapped message. This requirement makes the WSDL or its supporting schemas more complicated.
- In the document literal wrapped convention, the method name appears as a global schema element in the SOAP message. As per Schema rules, you cannot have more than one global schema element with the same name. So if there is a need to over

load an operation, which would require having two or more operations with the same name, you cannot use the document literal wrapped style.

In the example shown below, the input data for the operation getEmployeeInfo is contained in the <getEmployeeInfo> element. The output data, or response, from that same operation is contained within the < getEmployeeInfoResponse> element.

```
<soap:envelope>
   <soap:body>
    <tns:getEmployeeInfo>
      <tns:employeeInfoRequest>
         <tns1:employeeId>AXB001001</tns1:employeeId>
      </tns:employeeInfoRequest>
    </tns:getEmployeeInfo>
  </soap:body>
</soap:envelope>
```

Figure 25: Wire format of Document Literal Wrapped XML

(Source: WebSphere Integration Developer Docs)

Further details on document literal wrapped convention is provided in the Appendix A. We have discussed various best practices, next we will show how these best practices are applied with help of a case study.

4.0 How to Use – An Example

The objective of the following example is to show how best practices mentioned in the previous sections, can be applied to an example. We will be exploring a simple case study of an "employee service" which was designed based on top down approach. The process which is elaborated in Figure 26, was initiated by a business owner requesting the design of Employee Service.

1. Common practice is to use copybooks to arrive at XML Schema (Bottom up approach)
   - The downside is limited visibility of a broader solution applicable to multiple instance rather one single self contained instance. Thus the service interface / message specification will become less reusable is we start bottom up/
2. Our recommendation is to use a requirements driven, top-down approach for message one sec I am checking
3.
   - If we do not do top-down we will not be able to effectively utilize an enterprise messaging specification (EIMS).

Core data type: Data elements used across the enterprise (not specific to any domain)

Customer Name, Address,

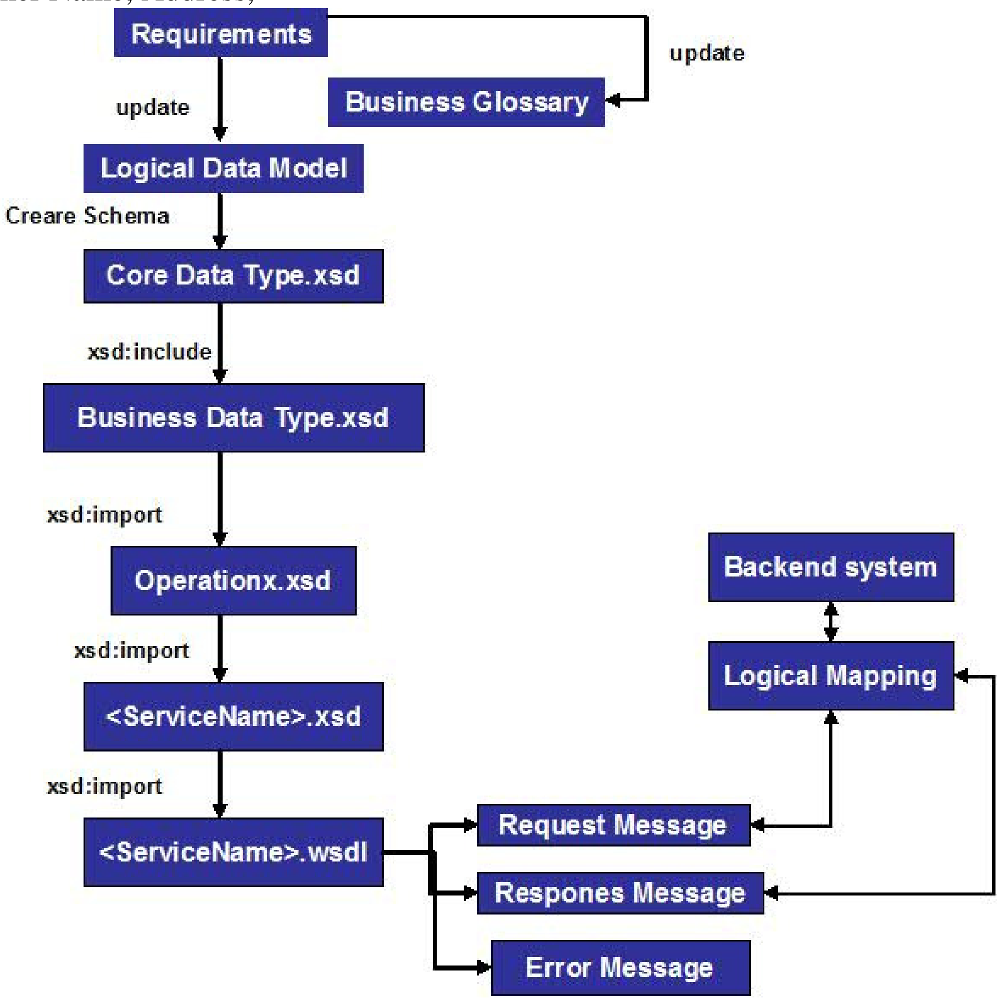


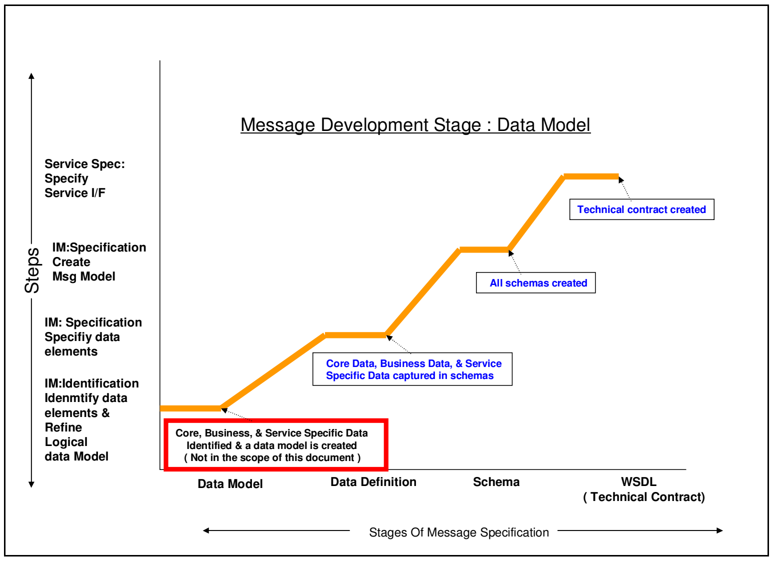
Figure 26: Top down development stages for case study.

The requirement was to retrieve employee details and organization information from a back end system based on employee identification number. One of the key requirements was that the service that was being developed be reusable across the enterprise.

As stated before, it is important to reemphasize that data is very important part of an Enterprise and its SOA solutions. As a physical realization of service contracts gets developed, it becomes important to look at data or sets of data which are needed to support all components of service contract. So evaluation of the data model for the service contract would be the first step.

4.1 Data Model

A data model is an abstract model that describes how data is represented and accessed. So the first stage of top down message development deals with the data and its dependencies. The objective here is to ensure all the business entities are analyzed and captured to ensure the service successfully compliments the business process it is trying to address.



In the context of employee service, we were able to identify four key business entities which are "organization", "employee", "name" and "address". These entities were then modeled to define their relationships with each other. The result of this exercise was the following model which shows the business entities and their relationship. What we have now is the identification and understanding of the business components required to support the service contract.

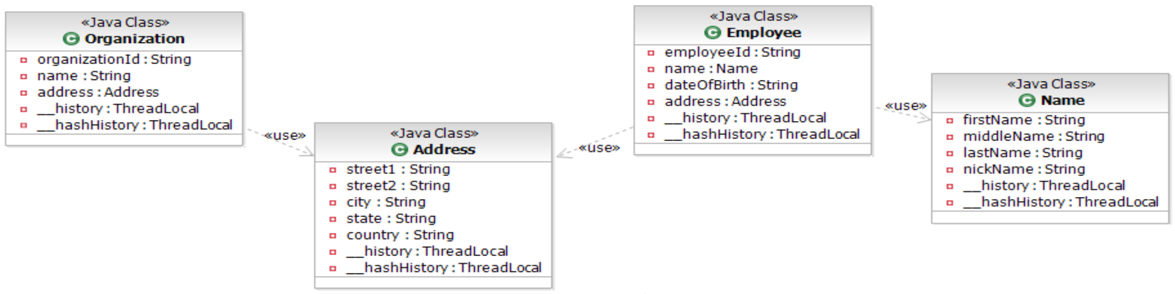
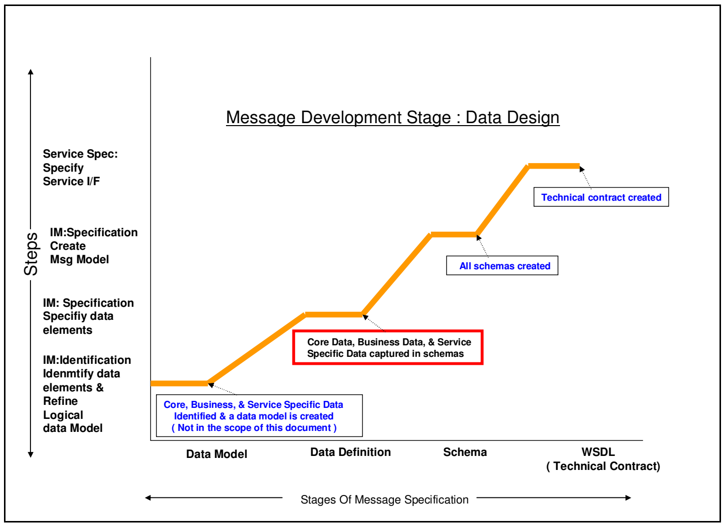


Figure 27: Logical Data Model

4.2 Data Definition

Now that the data model has been defined and the business entities have been captured, Message Specification moves to the data definition stage as shown in the adjacent message development stage figure. In this stage, based on the logical data model, we create schema constructs as complex and simple types.


Message Development Stage : Data Design
Steps
Service Spec: Specify Service I/F
IM:Specification Create Msg Model
IM: Specification Specifiy data elements
IM:Identification Idenmtify data elements & Refine Logical data Model
Technical contract created
All schemas created
Core Data, Business Data, & Service Specific Data captured in schemas
Core, Business, & Service Specific Data Identified & a data model is created (Not in the scope of this document)
Data Model
Data Definition
Schema
WSDL (Technical Contract)
Stages Of Message Specification

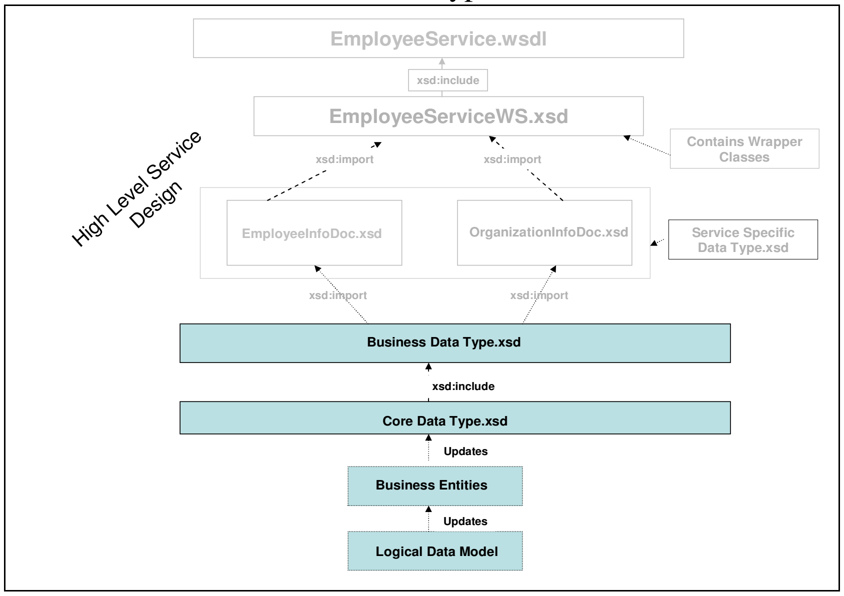
In our case study on Employee Service, we start by defining Core Data Type schema file which has all the simple data types defined. The definitions would range from string based simple types to integer based simple types as shown in Figure 28 below. The idea here is to have a core data type definition in a separate schema at the enterprise level which could be "reused" across projects and services.

```
<xsd:simpleType name=”Alpha2”>
        <xsd:restriction base="xsd:string">
                <xsd:maxLength value="2"/>
        </xsd:restriction>
</xsd:simpleType>
```

```
<xsd:simpleType name="UnsignedMax5">
        <xsd:restriction base="xsd:integer">
                <xsd:maxInclusive value="99999"/>
        </xsd:restriction>
</xsd:simpleType>
```

Figure 28: Core Data Types

After creation of core data types, a business data type schema is created which would capture all the business entities reusable across the multiple operations for this service as well as other service in a specific domain. The business data type schema would do an xsd:include of the Core Data Type xsd as show in the adjacent "High Level Service Design". This would make all the constructs in the core data type accessible to business data type. The business data type schema can be used as a reusable component for this domain. The idea is that other services in this domain should also be able to leverage the constructs in the schema.



In this top down design approach while using all the recommended best practices, we have also used data encapsulation below (refer to Figure 29 where complex types Name and Address were used). The complex type Employee has an element called name of type Name. Employee (due to data encapsulation) is completely unaware about the details of Name.

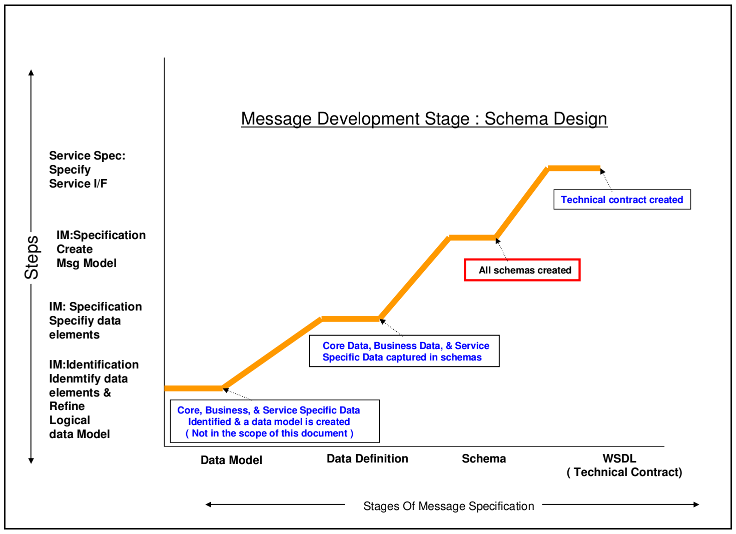
The following Figure 29 shows the business data type schema for the employee services. And also shows the best practices of good design. The schema constructs are readable and understandable by anyone visually looking at them. There is also good consistency in this schema ranging from naming of the types, schema structures that are used accompanied by logical organization of data. All of these make it easy to work with such schemas.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:tns="http://www.example.org/HumanResource/businessdatatype/"
                              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                              targetNamespace="http://www.example.org/HumanResource/businessdatatype/"
                              elementFormDefault="qualified" attributeFormDefault="unqualified">

        <!-- Including coreDataType with no namespace -->
        <xsd:include schemaLocation="CoreDataType.xsd"/>

       <xsd:element name="employee" type="tns:Employee"></xsd:element>
        <xsd:complexType name="Employee">
                <xsd:sequence>
                        <xsd:element name="employeeId" type="tns:Alpha20"></xsd:element>
                        <xsd:element name="name" type="tns:Name"></xsd:element>
                        <xsd:element name="dateOfBirth" type="tns:Alpha20"></xsd:element>
                        <xsd:element name="address" type="tns:Address"></xsd:element>
                <xsd:sequence>
        </xsd:complexType>

        <xsd:element name="name" type="tns:Name"></xsd:element>
        <xsd:complexType name="Name">
                <xsd:sequence>
                        <xsd:element name="firstName" type="tns:Alpha20"></xsd:element>
                        <xsd:element name="middleName" type="tns:Alpha20"></xsd:element>
                        <xsd:element name="lastName" type="tns:Alpha20"></xsd:element>
                        <xsd:element name="nickName" type="tns:Alpha20"></xsd:element>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:element name="organization" type="tns:Organization"></xsd:element>
        <xsd:complexType name="Organization">
                <xsd:sequence>
                        <xsd:element name="organizationId" type="tns:Alpha20"></xsd:element>
                        <xsd:element name="name" type="tns:Alpha20"></xsd:element>
                        <xsd:element name="address" type="tns:Address"></xsd:element>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:element name="address" type="tns:Address"></xsd:element>
        <xsd:complexType name="Address">
                <xsd:sequence>
                        <xsd:element name="street1" type="tns:Alpha60"></xsd:element>
                        <xsd:element name="street2" type="tns:Alpha60"></xsd:element>
                        <xsd:element name="city" type="tns:Alpha20"></xsd:element>
                        <xsd:element name="state" type="tns:Alpha20"></xsd:element>
                        <xsd:element name="country" type="tns:Alpha20"></xsd:element>
                </xsd:sequence>
        </xsd:complexType>
</xsd:schema>
```

Figure 29: Business Data Types Schema (BusinessDataType.xsd)

4.3 Schema

Now that the data definition has been completed and the business entities have been captured, Message Specification moves to the schema stage as pointed in the adjacent message development stage figure. In this stage, we start with the approach of operation based schema design. In context of EmployeeService, we have getEmployeeInfo and getOrganizationInfo operations defined in the service contract. These two operations give rise to two schemas – “OrganizationInfoDoc.xsd” and “EmployeeInfoDoc.xsd”. Each of the schemas contains the schemas constructs to support the individual operations in the service contract. Each operation specific schema does an import of the domain specific BusinessDataType.xsd schema. Thus we end up having the schemas dependent as shown in Figure 29.



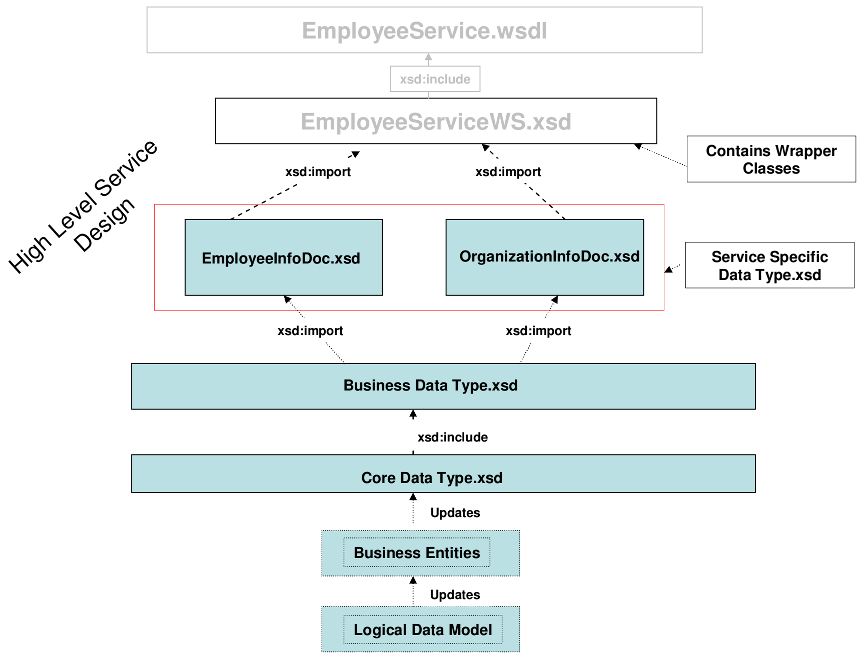


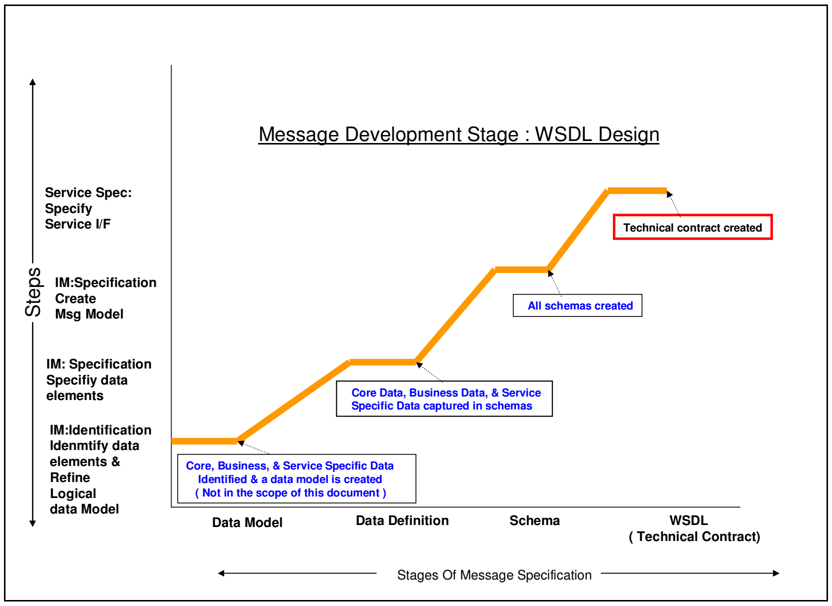
Figure 30: Operation Driven Schema Design.

The following figure shows in detail the OrganizationInfoDoc.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:data=http://www.example.org/HumanResource/businessdatatype/
   targetNamespace=
           "http://www.example.org/HumanResource/EmployeeService/"
   xmlns:tns="http://www.example.org/HumanResource/EmployeeService/"
   elementFormDefault="qualified">

   <xsd:import
    namespace="http://www.example.org/HumanResource/businessdatatype/"
    schemaLocation="BusinessDataType.xsd"/>

   <!-- These elements enable the document/literal/wrapped convention.
   -->
   <!-- These types define the structure of request and response
   interfaces. -->

   <xsd:element name="getOrganizationInfoRequest"
            type="tns:OrganizationInfoRequest" />
   <xsd:complexType name="OrganizationInfoRequest">
           <xsd:sequence>
                       <xsd:element name="organizationId" type="data:Alpha10"/>
           </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="getOrganizationInfoResponse"
            type="tns:OrganizationDetails"/>
   <xsd:complexType name="OrganizationDetails">
      <xsd:sequence>
                       <xsd:element name="organization" type="data:Organization"/>
            </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

Figure 31: Schema for getOrganizationInfo operation (OrganizationInfoDoc.xsd).

4.4 WSDL



Now that the operation based schema design has been completed, Message Specification moves to the WSDL stage as shown in the adjacent message development stage figure. The WSDL, as stated before, provides a standardized, complete service contract that potential consumers can use to access the service without knowledge of how the actual service logic is implemented. The first step of implementing the service contract is to create a schema file - “EmployeeServiceWS.xsd”. The objective of having this schema is to hold the wrapper data types required as part of document/literal wrapped convention. This schema file also ties in all the dependent schemas through the use of xsd:import to enable support for the service contract as shown in Figure 31.

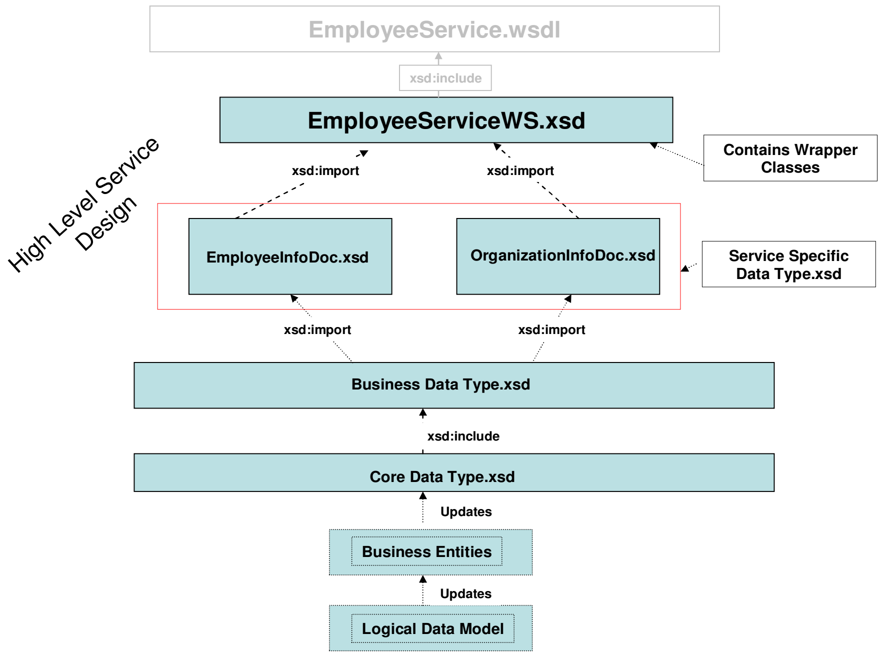


Figure 32: High Level Design including EmployeeServiceWS.xsd

Figure 33 below shows the EmployeeServiceWS.xsd and how it is used to tie the supporting schemas in respective name spaces to support the service contract for Employee Service.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:emp="http://www.example.org/HumanResource/EmployeeService/"
        xmlns:tns="http://www.example.org/ws/HumanResource/EmployeeService/"
        targetNamespace="http://www.example.org/ws/HumanResource/EmployeeService/"
        elementFormDefault="qualified">

        <xsd:import  namespace="http://www.example.org/HumanResource/EmployeeService/"
                   schemaLocation="EmployeeInfoDoc.xsd"/>
        <xsd:import  namespace="http://www.example.org/HumanResource/EmployeeService/"
                   schemaLocation="OrganizationInfoDoc.xsd"/>
        <xsd:include schemaLocation="EmployeeServiceFault.xsd"/>

        <xsd:element name="getEmployeeInfo">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="employeeInfoRequest"
                  type="emp:EmployeeInfoRequest"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="getEmployeeInfoResponse">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="employeeInfoResponse"
                  type="emp:EmployeeResponse"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="getOrganizationInfo">
                <xsd:complexType>
                        <xsd:sequence>
                        <xsd:element name="organizationInfoRequest" type="emp:OrganizationInfoRequest"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="getOrganizationInfoResponse">
                <xsd:complexType>
                        <xsd:sequence>
                        <xsd:element name="organizationInfoResponse" type="emp:OrganizationDetails"/>
                        </xsd:sequence>
                </xsd:complexType>
        </xsd:element>
        <xsd:element name="fault" type="tns:Fault"/>
</xsd:schema>
```

Figure 33: Schema for EmployeeServiceWS.xsd

Once the wrapper schema file EmployeeServiceWS.xsd is completed, the service contract is created with include of EmployeeServiceWS.xsd in the wsdl:types section as shown in Figure 33 below.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://www.example.org/ws/HumanResource/EmployeeService/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  name="EmployeeService"
                  targetNamespace="http://www.example.org/ws/HumanResource/EmployeeService/">

         <wsdl:types>
                  <xsd:schema
                   targetNamespace="http://www.example.org/ws/HumanResource/EmployeeService/"
                    xmlns:tns="http://www.example.org/ws/HumanResource/EmployeeService/"
                    xmlns="http://www.w3.org/2001/XMLSchema"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
                    <xsd:include schemaLocation="EmployeeServiceWS.xsd"/>
             </xsd:schema>
         </wsdl:types>

         <wsdl:message name="getEmployeeInfoRequest">
                    <wsdl:part element="tns:getEmployeeInfo" name="parameters"/>
         </wsdl:message>
         <wsdl:message name="getEmployeeInfoResponse">
                    <wsdl:part element="tns:getEmployeeInfoResponse" name="parameters"/>
         </wsdl:message>
         <wsdl:message name="getOrganizationInfoRequest">
                  <wsdl:part name="parameters" element="tns:getOrganizationInfo"></wsdl:part>
         </wsdl:message>
         <wsdl:message name="getOrganizationInfoResponse">
                  <wsdl:part name="parameters"   element="tns:getOrganizationInfoResponse"></wsdl:part>
         </wsdl:message>

         <wsdl:message name="faultMsg">
                  <wsdl:part name="fault" element="tns:fault"/>
         </wsdl:message>

         <wsdl:portType name="EmployeeService">
                  <wsdl:operation name="getEmployeeInfo">
                     <wsdl:input message="tns:getEmployeeInfoRequest"/>
                     <wsdl:output message="tns:getEmployeeInfoResponse"/>
                     <wsdl:fault name="fault" message="tns:faultMsg"/>
                   </wsdl:operation>

                   <wsdl:operation name="getOrganizationInfo">
                     <wsdl:input message="tns:getOrganizationInfoRequest"></wsdl:input>
                     <wsdl:output message="tns:getOrganizationInfoResponse"></wsdl:output>
                     <wsdl:fault name="fault" message="tns:faultMsg"/>
                    </wsdl:operation>
         </wsdl:portType>

         <wsdl:binding name="EmployeeServiceSOAP" type="tns:EmployeeService">
                    <soap:binding style="document"
                    transport="http://schemas.xmlsoap.org/soap/http"/>
                      <wsdl:operation name="getEmployeeInfo">
                        <soap:operation
            soapAction=
            "http://www.example.org/EmployeeService/getEmployeeInfo/V1/"/>
                        <wsdl:input>
                          <soap:body use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                          <soap:body use="literal"/>
                        </wsdl:output>
                        <wsdl:fault name="fault">
                                <soap:fault name="fault" use="literal"/>
                        </wsdl:fault>
                      </wsdl:operation>

                      <wsdl:operation name="getOrganizationInfo">
                        <soap:operation soapAction=
           "http://www.example.org/EmployeeService/getOrganizationInfo/V1/"/>
                        <wsdl:input>
                          <soap:body use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                          <soap:body use="literal"/>
                        </wsdl:output>
                                <wsdl:fault name="fault">
                                   <soap:fault name="fault" use="literal"/>
                                </wsdl:fault>
                      </wsdl:operation>
                    </wsdl:binding>

                    <wsdl:service name="EmployeeService">
                                <wsdl:port binding="tns:EmployeeServiceSOAP"
                    name="EmployeeServiceSOAP">
                                        <soap:address
                    location="http://www.example.org/EmployeeService"/>
                                </wsdl:port>
                    </wsdl:service>
</wsdl:definitions>
```

Figure 34: EmployeeService.wsdl

4.5 Request and Response Messages

In previous sections, we examined best practices for message specification. In this section we will create the request and response messages for the respective operations and the associated attributes.

Some of the features of the request and response messages are:

- The input message has a single part.
- The part is an element.
- The element has the same name as the operation.
- The element's complex type has no attributes.

Figures 33, 34, 35 and 36 shows the request and response messages for the getEmployeeInfo and getOrganizationInfo operation.

```
<tns:getEmployeeInfo
           xmlns:tns="http://www.example.org/HumanResource/EmployeeService/"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation=
"http://www.example.org/ws/HumanResource/EmployeeService/ EmployeeServiceWS.xsd
http://www.example.org/HumanResource/EmployeeService/ EmployeeInfoDoc.xsd
http://www.example.org/HumanResource/EmployeeService/ OrganizationInfoDoc.xsd ">
  <employeeInfoRequest>
    <tns:employeeId>00123221</tns:employeeId>
  </employeeInfoRequest>
</tns:getEmployeeInfo>
```

Figure 35: GetEmployeeInfo: Request Message

Since, we are using document literal style and wrapped convention, everything that appears in soap:body is defined by the schema. This provides for easy validation of the message. The method name also appears in the SOAP message. Also in this case for getEmployeeInfo and getOrganizationInfo there is no type encoding information. Document literal is WS-I compliant, and the wrapped pattern meets the WS-I restriction that the SOAP message's soap:body has only one child.

```
<tns:getEmployeeInfoResponse
      xmlns:tns="http://www.example.org/HumanResource/EmployeeService/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.example.org/ws/HumanResource/EmployeeService/
EmployeeServiceWS.xsd http://www.example.org/HumanResource/EmployeeService/
EmployeeInfoDoc.xsd http://www.example.org/HumanResource/EmployeeService/
OrganizationInfoDoc.xsd ">
  <employeeInfoResponse>
    <tns:employee>
      <employeeId>00123221</employeeId>
      <name>
```

```
        <firstName>Daffy</firstName>
        <middleName>Collins</middleName>
        <lastName>Duck</lastName>
        <nickName>Daffy</nickName>
      </name>
      <dateOfBirth>dateOfBirth</dateOfBirth>
      <address>
        <street1>1101 Acme St</street1>
        <street2>#101</street2>
        <city>Acme City</city>
        <state>CA</state>
        <country>USA</country>
      </address>
    </tns:employee>
    <tns:organization>
      <organizationId>AS223223</organizationId>
      <name>name</name>
      <address>
        <street1>2102 Acme St</street1>
        <street2>#101</street2>
        <city>Acme City</city>
        <state>CA</state>
        <country>USA</country>
      </address>
    </tns:organization>
  </employeeInfoResponse>
</tns:getEmployeeInfoResponse>
```

Figure 36: GetEmployeeInfo: Response Message

```
<tns:getOrganizationInfo
      xmlns:tns="http://www.example.org/HumanResource/EmployeeService/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.example.org/ws/HumanResource/EmployeeService/
EmployeeServiceWS.xsd http://www.example.org/HumanResource/EmployeeService/
EmployeeInfoDoc.xsd http://www.example.org/HumanResource/EmployeeService/
OrganizationInfoDoc.xsd ">
  <organizationInfoRequest>
    <tns:organizationId> AS223223</tns:organizationId>
  </organizationInfoRequest>
</tns:getOrganizationInfo>
```

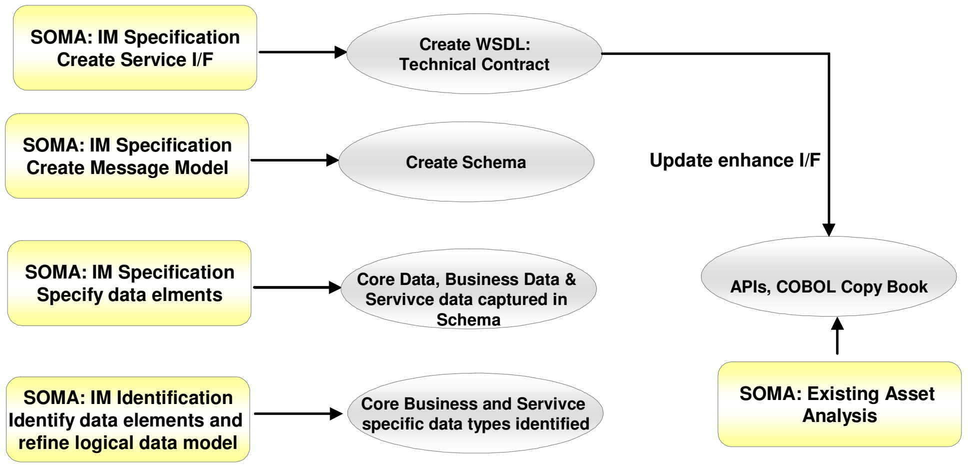
Figure 37: GetOrganizationInfo: Request Message

```
<tns:getOrganizationInfoResponse
      xmlns:tns="http://www.example.org/HumanResource/EmployeeService/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.example.org/ws/HumanResource/EmployeeService/
EmployeeServiceWS.xsd http://www.example.org/HumanResource/EmployeeService/
EmployeeInfoDoc.xsd http://www.example.org/HumanResource/EmployeeService/
OrganizationInfoDoc.xsd ">
  <organizationInfoResponse>
    <tns:organization>
      <organizationId>AS223223</organizationId>
      <name>name</name>
      <address>
        <street1>2102 Acme St</street1>
        <street2>#101</street2>
        <city>Acme City</city>
        <state>CA</state>
        <country>USA</country>
      </address>
    </tns:organization>
  </organizationInfoResponse>
</tns:getOrganizationInfoResponse>
```

Figure 38: GetOrganizationInfo: Response Message

5.0 Usage

Summary of SOMA steps involved in message specification is shown below.


SOMA: IM Specification Create Service I/F
Create WSDL: Technical Contract
SOMA: IM Specification Create Message Model
Create Schema
Update enhance I/F
SOMA: IM Specification Specify data elments
Core Data, Business Data & Servivce data captured in Schema
APIs, COBOL Copy Book
SOMA: IM Identification Identify data elements and refine logical data model
Core Business and Servivce specific data types identified
SOMA: Existing Asset Analysis

Message specification is initiated during Identification phase and performed during the Service Specification phase. The purpose of this activity is to identify the service interface and associated request and response messages. The recommended approach is to use a Top down Design Approach which starts with business requirements from uses cases and business processes. If Bottom-up approach is used where e.g. COBOL copy book is used to create a service interface, then the chances are very high this interface will not be re-usable across the business domains. Once we have created the service interface using the top down approach (business requirement drive approach), we should use this interface to enhance the COBOL copy books and other APIs of existing assets to ensure consistency and compatibility between the back-end interfaces and the service interface.

We discussed best practices all the above sections, we have also provided additional best practices in Appendix B.. We showed how these best practices can be applied with the help of an example in section 4.

6.0 Key Considerations

Message specification should be driven by business requirements if the service is to be re-usable across business areas. Bottom-up approach to create service interfaces based on existing code or Cobol copy book would lead to brittle interfaces.

7.0 Estimation Guidelines

The estimation below is based on complexity of service involved. It is for each service.

Message Specifica Tasks	Low Effort Estimation	Medium Effort Estimation	High Effort Estimation
Data Model: Identification of business entities and their relationships. Note: This task would be easier if Conceptual and Logical data nmodel exist	40 hours	80 hours	200 hours
Data Definition: Capture business entities in the Schemas, specify data types	8 hours	20 hours	40 hours
Schema: Design operation level schemas, link them with core and bsuiness data type schemas	8 hours	20 hours	40 hours
WSDL: Create service contract	8 hours	16 hours	24 hours

8.0 Related Techniques

The related techniques are Information Modeling, Domain Decomposition.

9.0 Roles, Input and Output Work products

Roles: Information Architect, SOA Architect,

Input Work products: Information Model, Use Cases, Business Process Model, Logical Data Model

Appendix A: Wrapped document/literal WSDL

Russell Butek (butek@us.ibm.com), Developer, IBM

http://www.ibm.com/developerworks/webservices/library/ws-tip-namespace.html#N10155

The WSDL in Listing 5 is the equivalent to the WSDL in Listing 1. It is document/literal wrapped instead of RPC/literal. The Java APIs generated for this WSDL are identical to the Java APIs generated for the RPC/literal WSDL, but the SOAP messages are potentially somewhat different. Once again, the namespaces are highlighted in bold.

Listing 5. document/literal wrapped WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    targetNamespace="http://apiNamespace.com"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://apiNamespace.com"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/">
  <types>
    <schema
        targetNamespace="http://refNamespace.com"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://refNamespace.com">
      <element name="RefDataElem" type="int"/>
    </schema>
    <schema
        targetNamespace="http://dataNamespace.com"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:ref="http://refNamespace.com"
        xmlns:tns="http://dataNamespace.com"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <import namespace="http://refNamespace.com"/>
      <complexType name="Data">
        <sequence>
          <element name="data1" type="int"/>
          <element name="data2" type="int"/>
        </sequence>
      </complexType>
      <element name="DataElem" nillable="true" type="tns:Data"/>
      <complexType name="Data2">
        <sequence>
          <element ref="ref:RefDataElem"/>
        </sequence>
      </complexType>
    </schema>
    <schema
        targetNamespace="http://apiNamespace.com"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:data="http://dataNamespace.com"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <import namespace="http://dataNamespace.com"/>
      <element name="op1">
        <complexType>
          <sequence>
            <element name="in" type="data:Data"/>
          </sequence>
        </complexType>
      </element>
      <element name="op1Response">
        <complexType>
          <sequence>
            <element name="op1Return" type="data:Data"/>
          </sequence>
        </complexType>
      </element>
      <element name="op2">
        <complexType>
          <sequence>
            <element name="in" type="data:Data"/>
          </sequence>
        </complexType>
      </element>
      <element name="op2Response">
        <complexType>
          <sequence>
            <element name="op2Return" type="data:Data"/>
          </sequence>
        </complexType>
      </element>
      <element name="op3">
        <complexType>
          <sequence>
            <element ref="data:DataElem"/>
            <element name="in2" type="data:Data2"/>
          </sequence>
        </complexType>
      </element>
      <element name="op3Response">
        <complexType>
          <sequence>
            <element name="op3Return" type="data:Data2"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </types>
  <message name="op1Request">
    <part element="tns:op1" name="parameters"/>
  </message>
  <message name="op1Response">
    <part element="tns:op1Response" name="parameters"/>
  </message>
  <message name="op2Request">
    <part element="tns:op2" name="parameters"/>
  </message>
  <message name="op2Response">
```

```
    <part element="tns:op2Response" name="parameters"/>
  </message>
  <message name="op3Request">
    <part element="tns:op3" name="parameters"/>
  </message>
  <message name="op3Response">
    <part element="tns:op3Response" name="parameters"/>
  </message>
  <portType name="Sample">
    <operation name="op1">
      <input message="tns:op1Request"/>
      <output message="tns:op1Response"/>
    </operation>
    <operation name="op2">
      <input message="tns:op2Request"/>
      <output message="tns:op2Response"/>
    </operation>
    <operation name="op3">
      <input message="tns:op3Request"/>
      <output message="tns:op3Response"/>
    </operation>
  </portType>
  <binding name="SampleSoapBinding" type="tns:Sample">
    <wsdlsoap:binding style="document" transport=
    "http://schemas.xmlsoap.org/soap/http"/>
    <operation name="op1">
      <wsdlsoap:operation soapAction=""/>
      <input>
        <wsdlsoap:body namespace=
        "http://apiNamespace.com" use="literal"/>
      </input>
      <output>
        <wsdlsoap:body namespace=
        "http://apiNamespace.com" use="literal"/>
      </output>
    </operation>
    <operation name="op2">
      <wsdlsoap:operation soapAction=""/>
      <input>
        <wsdlsoap:body namespace=
        "http://op2Namespace.com" use="literal"/>
      </input>
      <output>
        <wsdlsoap:body namespace=
        "http://op2Namespace.com" use="literal"/>
      </output>
    </operation>
    <operation name="op3">
      <wsdlsoap:operation soapAction=""/>
      <input>
        <wsdlsoap:body use="literal"/>
      </input>
      <output>
        <wsdlsoap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="SampleService">
    <port binding="tns:SampleSoapBinding" name="Sample">
      <wsdlsoap:address location=
      "http://localhost:9080/Wrapped/services/Sample"/>
    </port>
  </service>
</definitions>
```

For op1, the SOAP messages for the document/literal wrapped service are identical to the SOAP messages for the RPC/literal service (see Listing 2). This is the most common sort of document/literal wrapped operation. Be aware that you're now following rule 2.1, not rule 1.1; you're getting the namespace from an element, not from a `wsdl:soapbody`. Notice the convention in the WSDL that the wrapper elements are defined in the same namespace as the WSDL itself: "http://apiNamespace.com." They could have been defined in any namespace, but by following this convention, the wrapped messages are identical to the RPC/literal messages.

op2's document/literal wrapped SOAP messages are in Listing 6.

Listing 6. Document/literal wrapped request/response SOAP messages for op2

```
<soapenv:Envelope xmlns:soapenv=
                       "http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <p582:op2 xmlns:p582="http://apiNamespace.com">
      <in>
        <data1>3</data1>
        <data2>4</data2>
      </in>
    </q0:op2>
  </soapenv:Body>
</soapenv:Envelope>

<soapenv:Envelope xmlns:soapenv=
"http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <p582:op2Response xmlns:p582="http://apiNamespace.com">
      <op2Return>
        <data1>300</data1>
        <data2>400</data2>
      </op2Return>
    </p582:op2Response>
  </soapenv:Body>
</soapenv:Envelope>
```

WS-I compliance

WS-I requires that document/literal bindings **do not** have a namespace attribute in their `wsdl:soapbody`. So op1 and op2 are non-compliant. I ran this in IBM® WebSphere® Studio Application Developer, which accepts non-compliant WSDL, but enforces this WS-I mandate by ignoring any `wsdl:soapbody` namespaces.

The document/literal wrapped messages for op2 make it much more obvious that you're following rule 2.1 rather than rule 1.1. You're completely ignoring the namespace in the binding's `wsdl:soapbody`.

Finally, let's compare the SOAP messages for the RPC/literal op3 (Listing 4) and the document/literal wrapped op3 (Listing 7). Like op2, the only difference is the namespace of op3. The RPC version has no namespace, but the document version does have a namespace; just like all the other document messages, it gets the namespace from the element's namespace: rule 2.1.

Listing 7. Document/literal wrapped request/response SOAP messages for op3

```
<soapenv:Envelope xmlns:soapenv=
                       "http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <p582:op3 xmlns:p582="http://apiNamespace.com">
      <p10:DataElem xmlns:p10="http://dataNamespace.com">
        <data1>5</data1>
        <data2>6</data2>
      </p10:DataElem>
      <in2>
        <p971:RefDataElem xmlns:p971=
        "http://refNamespace.com">7</p971:RefDataElem>
      </in2>
    </p582:op3>
  </soapenv:Body>
</soapenv:Envelope>

<soapenv:Envelope xmlns:soapenv=
"http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <p582:op3Response xmlns:p582="http://apiNamespace.com">
      <op3Return>
        <p971:RefDataElem xmlns:p971=
        "http://refNamespace.com">7</p971:RefDataElem>
      </op3Return>
    </p582:op3Response>
  </soapenv:Body>
</soapenv:Envelope>
```

Appendix B: Message Specification Best Practices and Architectural Decisions

SOMA Specification: Message Specification Best Practices - Use standard message format (e.g., SOAP) for Request, Response and Error messages

Current Situation:
Current message format such as message header format, message body format etc., do not conform to Web Services (WS) messaging standards
Recommended Approach:
WS Standard message format based on SOAP should be used for application to application communications across the enterprise.
Benefits:
• Standard format for Header, Body and Error elements would ensure consistent messaging structure across the enterprise • Provides for standardized implementation of security, routing, transaction co-ordination, error logging etc., across application domains • Reduces the need to build security, routing and transaction co-ordination capabilities in each application – need to leverage standard capabilities available in products

SOMA Specification: Message Specification Best Practices - Use standard message descriptions (e.g., WSDL) to describe the service

Current Situation:
Services are not described in standard format such as WSDL Web Services standard
Recommended Approach:
Services will follow a standard based message description for service publishing, discovery and invocation
Benefits:
• Allows for dynamic services • Can leverage tools to generate code from WSDL • Results in separation of concern i.e. business logic segregated from discovery and routing logic • Part of the required set of WS standard protocols – SOAP, WSDL, etc

Yes

SOMA – Specification: Message Specification Best Practices - Use web services standard security (e.g., WS-SECURITY) mechanisms to pass security credentials as part of the header

Current Situation:
Current messages pass security credentials along with application data
Recommended Approach:
Security credentials and encryption requirements will be handled by standard based integration components across the enterprise
Benefits:
• Reduces security credential exposure to application logic • Standardized service security framework across enterprise • Reduces application logic overhead – as security would be supported through handlers • Improves options on selecting level of security – header level, transport level or message body level, tokens instead of userid and password

SOMA Specification: Message Specification Best Practices: Use web services standard fault mechanisms (e.g., SOAP Fault) to communicate the error messages

Current Situation:
Message passes error information along with application data
Situation After:
Error information should be transmitted using standard fault mechanisms when needed
Benefits:
• Flexibility to handle errors in different layers such as message integration layer, message routing layer, business logic layer etc., • Standardized error formatting and handling across enterprise