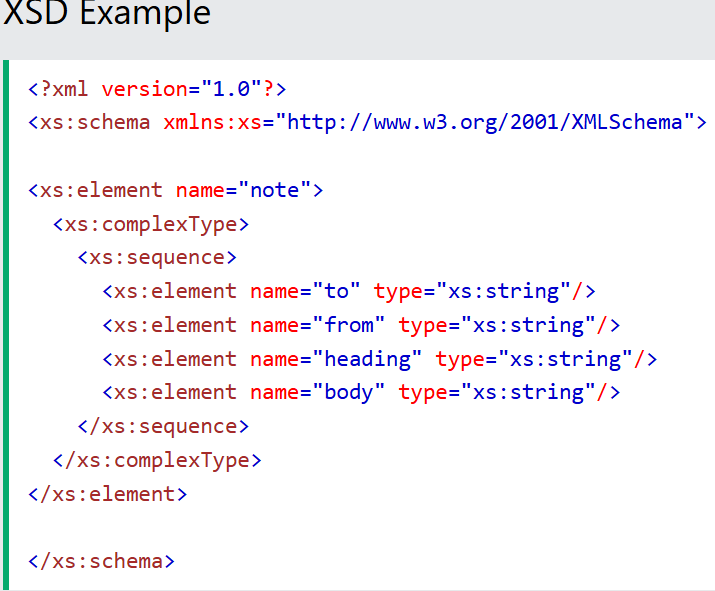
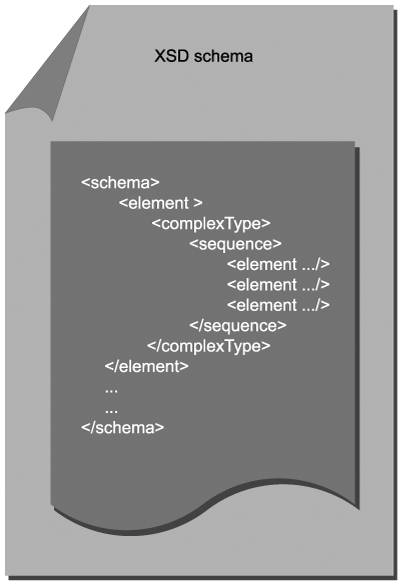
**XML Schema Definition Language**

The XML Schema Definition Language (XSD) has become a central and very common part of XML and Web services architectures. The hierarchical structure of XML documents can be formally defined by creating an XSD schemahence an XML document is considered an instance of its corresponding schema. Further, the structure established within an XSD schema (Figure 13.3) contains a series of rules and constraints to which XML document instances must comply for parsers and processors to deem them valid.

For details of XSD : https://www.w3schools.com/xml/schema\_intro.asp





The fundamental data representation rules provided by XSD schemas are related to representing data according to type. As with data types used in programming languages, XSD schemas provide a set of non-proprietary data types used to represent information in XML document instances.

The data types supported by XSD schemas are extensive, but they do not always map cleanly to the proprietary types used by programming languages. Therefore, many environments must go through a mapping process to select the XSD schema types best suited to represent data originating from or being delivered to proprietary application logic and data sources.

XSD schemas can exist as separate documents (typically identified by .xsd file extensions), or they may be embedded in other types of documents, such as XML document instances or WSDL definitions. XML document instances commonly link to a separate XSD schema file so that the same schema can be used to validate multiple document instances. WSDL definitions can import the contents of an XSD file, or they also can have schema content embedded inline.

Because almost all XML and Web services specifications are themselves written in XML, XSD schemas have become an intrinsic part of XML data representation and service-oriented architectures. Regardless of whether you explicitly define an XSD schema for your solution, your underlying processors will be using XSD schemas to execute many tasks related to the processing of XML documents through Web services. (The role of XSD schemas within SOA is explained further in the XML Schema and SOA section in Chapter 14.)

"Elements" vs. "Constructs"

Each of the specifications we explore in this and subsequent chapters provides a markup language that is expressed as an XML dialect. This means that the language itself is written in XML and is comprised of XML elements. Our focus is on describing key elements that provide features relevant to the topics discussed in this book. Sometimes we refer to language elements as constructs. A construct simply represents a key parent element likely to contain a set of related child elements. Therefore we use this term to communicate that the element we are discussing will contain additional elements that form a block of XML.

13.2.1. The schema element

The schema element is the root element of every XSD schema. It contains a series of common attributes used primarily to establish important namespace references. The xmlns attribute, for example, establishes a default namespace on its own, or can be used to declare a prefix that acts as an identifier to qualify other elements in the schema.

The http://www.w3.org/2001/XMLSchema namespace always is present so that it can be used to represent content in the schema that originates from the XSD specification and the elements in the schema document itself. This allows processors to distinguish native XSD schema content from user-defined content.

Other important attributes include **targetNamespace**, used to assign a namespace to the custom elements and attributes declared in the schema document, and the element-FormDefault attribute, which when set to a value of "qualified," requires that all elements in the XML document be associated with their corresponding namespace.

Using this element, you can declare a custom element that is then referenced by its name within an XML document instance.

Example 13.3. The usage of this element in an XML document instance.

12345

...where the value in between the opening and closing InvoiceNumber tags is required to be an integer.

The type attribute of an element can be set to one of the predefined data types established by the XML Schema specification, or it can be assigned a complex type, as explained next.

13.2.3. **The complexType and simpleType elements**

With a complexType you can group elements and attributes into a composite type that can be used to represent a set of related data representations. The following example groups two elements named ID and WeeklyHoursLimit into a complexType named EmployeeHours.

Example 13.4. A complexType containing two element declarations.

<complexType name="EmployeeHours">

complexType>

The EmployeeHours complexType can be assigned to one or more elements. This facilitates standardization and reuse of commonly grouped information and avoids redundant element declarations. Note that the sequence element is a type of indicator used within the complexType construct to establish a specific order for element elements.

simpleType elements also allow you to group related data representations, but these constructs cannot contain attributes or further child elements. (None of the examples used in this book contain simpleType elements.)

13.2.4. **The import and include elements**

XSD schemas can be modularized. This allows for one schema document to import the contents of another. Both the import and include elements are used to point to the location of the XSD schema file that will be pulled in when the schema is processed at runtime.

Example 13.5. The import and include elements.

<import schemaLocation="..."

namespace="http://www.xmltc.com/tls/schema"/>

<include schemaLocation="..."/>

The difference between these two elements is that include is used to reference schemas that use the same target namespace as the parent schema, whereas import is used to point to schemas that use a different target namespace. As per the previous example, a namespace attribute only is used with the import element.

13.2.5. Other important elements

The XML Schema Definition Language is large and complex and provides numerous options for structuring and validating XML document data. There are many other important parts of the language that are not used in the examples provided in this book, including:

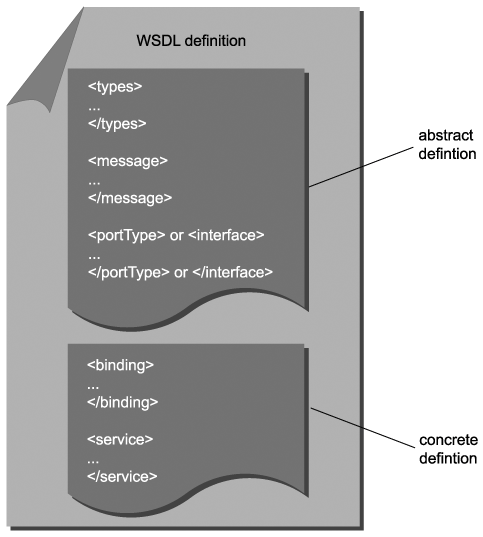
* additional type definition elements (attribute, complexContent, simpleContent)
* constraint related elements (restriction, enumeration, pattern)
* element indicators (maxOccurs, minOccurs, group)
* extensibility elements (any, extension, redefine)
* elements for simulating relationships between elements (unique, key, keyref)

**WSDL language basics**

The Web Services Description Language (WSDL) is the most fundamental technology standard associated with the design of services.

Just to recap, the abstract definition contains a series of parts that include types, message, and port type (or interface), whereas the concrete definition is comprised of binding and service parts. Each of these parts relates to corresponding elements (Figure 13.4) that are defined in the WSDL specification.

Figure 13.4. The structure of a WSDL definition.



13.3.1. The definitions element

This is the root or parent element of every WSDL document. It houses all other parts of the service definition and is also the location in which the many namespaces used within WSDL documents are established.

Example 13.6. A definitions element of the Employee Service, declaring a number of namespaces.

<definitions name="Employee"

targetNamespace="http://www.xmltc.com/tls/employee/wsdl/"

xmlns="http://schemas.xmlsoap.org/wsdl/"

xmlns:act="http://www.xmltc.com/tls/employee/schema/accounting/"

xmlns:hr="http://www.xmltc.com/tls/employee/schema/hr/"

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

xmlns:tns="http://www.xmltc.com/tls/employee/wsdl/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema">

...

definitions>

In the preceding example, the service definition is started with a definitions element that contains a series of attributes in which the service is assigned the name of "Employee" and in which a number of namespaces are declared.

For example, the xmlns attribute establishes the standard value of http://schemas.xmlsoap.org/wsdl/ as the default namespace. This means that all of the elements that belong to the WSDL language do not need prefixes to associate them with the WSDL specification.

By defining the xmlns:xsd namespace declaration, all elements within the WSDL that belong to the XML Schema Definition Language need to be prefixed with the xsd: qualifier. Also note the use of the xmlns:act and xmlns:hr namespace declarations. These are used to distinguish between two separate schemas that are imported into the types construct.

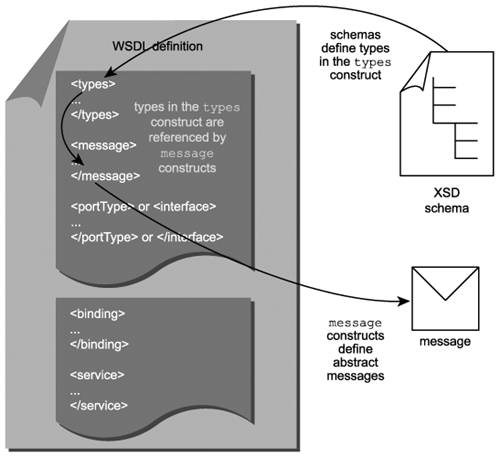
The xmlns:soap namespace declaration establishes the soap: qualifier used by elements introduced later in the bindings construct, where the WSDL definition associates its abstract operations to concrete SOAP bindings.

13.3.2. The types element

The types construct is where XSD schema content is placed. This part of the WSDL can consist of actual XSD schema markup (an entire schema construct containing type definitions), or it can contain import elements that reference external schema definitions (or it can contain both embedded and imported XSD types).

As illustrated in Figure 13.5, the types established in this part of the WSDL definition are used to represent the XML content of message bodies. The message element (explained later) references these types and associates them with messages.

Figure 13.5. The WSDL types construct populated by XSD schema types used by the message construct to represent the SOAP message body.



The SOAP message body contains XML content that can represent anything from simple parameter data to complex business documents. This content can be formally defined through types provided by the WSDL types area. Therefore, XSD schema complexType elements are commonly provided here, as they consist of groups of related types that can represent entire message body structures.

In the following example, an entire schema construct is embedded within the WSDL types construct.

Example 13.7. A types construct containing an XSD schema construct in which a complexType is defined.

<types>

types>

Use of the types construct is common in WSDL definitions with substantial content. However, it is not a required element. Native XSD schema types can be referenced directly within the message element, as explained next.

13.3.3. The message and part elements

For every message a service is designed to receive or transmit, a message construct must be added. This element assigns the message a name and contains one or more part child elements that each are assigned a type. message elements later are associated to operation elements to establish the input and output messages of the operation.

part elements use the type or element attributes to identify the data type of the message part. The type attribute can be assigned a simple or complex type and generally is used for RPC-style messages. part elements in document-style messages typically rely on the element attribute, which can reference an XSD element element. The name attribute is used to uniquely identify part elements within a message construct.

In the example below, we define request and response messages with two separate message constructs. Each message contains a part element that is assigned a predefined XSD element using the element attribute.

Example 13.8. Two message constructs likely representing the input and output messages for an operation.

<message name="getEmployeeWeeklyHoursRequestMessage">

<part name="RequestParameter"

element="act:EmployeeHoursRequestType"/>

message>

<message name="getEmployeeWeeklyHoursResponseMessage">

<part name="ResponseParameter"

element="act:EmployeeHoursResponseType"/>

message>

In the next example the part element is simply assigned a native XSD schema data type using the type attribute.

Example 13.9. A simple parameter message requiring just a single integer value.

type="xsd:integer"/>

If all messages in a WSDL definition are assigned native (simple) XSD types, a separate types construct generally is not required.

13.3.4. The portType, interface, and operation elements

Service operations are defined within the portType area of the WSDL definition. Hence, portType constructs simply represent collections of operations. Individual operations are defined using the aptly named operation element.

Example 13.10. The portType construct hosting two operation constructs.

<portType name="EmployeeInterface">

<operation name="GetWeeklyHoursLimit">

...

operation>

<operation name="UpdateHistory">

...

operation>

portType>

The portType element is defined in version 1.1 of the Web Services Description Language. Version 2.0 of this specification changes the name of this element to interface. The examples provided in this book are based on WSDL 1.1 and therefore continue to use the portType element.

13.3.5. The input and output elements (when used with operation)

Each operation construct contains input and/or output child elements that represent the request and response messages the operation is capable of processing.

In the example below, each operation has one input and one output message. The respective input and output elements are assigned messages defined in the message constructs (explained in the previous section) via their message attributes.

Example 13.11. operation elements with child input and output elements.

<input message="tns:getWeeklyHoursRequestMessage"/>

<output message="tns:getWeeklyHoursResponseMessage"/>

<input message="tns:updateHistoryRequestMessage"/>

<output message="tns:updateHistoryResponseMessage"/>

WSDL supports predefined message exchange patterns (MEPs). The presence of input and output elements and the sequence in which they are displayed generally establishes the MEP of the operation. For instance, the two operations defined in the previous example represent the request-response MEP.

Placing a single input element within the operation construct expresses the one-way MEP, as shown in the next example.

Example 13.12. An operation element with a single child input element.

<input message="tns:receiveSubmitMessage"/>

Note

It may seem confusing to associate request-and-response with a sequence of input and then output messages because there is a tendency to think that a request requires the service to initiate communication. The reason this makes sense in the Web services world is because WSDL definitions express an interface from the perspective of the service provider. So the request-response MEP, to a WSDL, means that a requestor will send it (the service provider) a request as input, to which it (the service provider) will reply with a response as output.

13.3.6. The binding element

So far, all of the elements we've described belong to the abstract service definition. On their own, they complete the description of a service interfacebut without referencing any means of messaging communication technology.

**The binding element begins the concrete portion of the service definition, to assign a communications protocol that can be used to access and interact with the WSDL.**

Upon first glance of the following example, the binding element appears similar in structure to the portType element. As with portType, the binding construct contains one or more operation elements. However, you'll notice the additional soap:binding and soap:operation elements interspersed within the construct syntax. These are what establish the SOAP protocol as the manner in which this WSDL can be communicated with.

Example 13.13. The binding construct hosting concrete operation definitions.

<binding name="EmployeeBinding" type="tns:EmployeeInterface">

<soap:binding style="document"

transport="http://schemas.xmlsoap.org/soap/http"/>

<soap:operation soapAction="..."/>

...

<soap:operation soapAction="..."/>

...

binding>

Further, the style attribute of the soap:binding element defines whether the SOAP messages used to support an operation are to be formatted as document or RPC-style messages.

The value of "document" allows the SOAP message body to contain a fully definable XML document structure. Assigning a value of "rpc" to the style attribute requires compliance to a body structure defined within the SOAP specification, which primarily forces the root element of the body to be named after the operation name.

13.3.7. The input and output elements (when used with binding)

Each operation element within a binding construct mirrors the input and output message child elements defined in the abstract definition.

Within a binding construct, however, the input and output elements do not reference the message elements again. Instead, they contain protocol details that establish how the messages are going to be processed and interpreted by the chosen communication technology. In our example, the service interface has been bound to the SOAP protocol.

Example 13.14. input and output elements providing message processing information.

<input>

<soap:body use="literal"/>

input>

<output>

<soap:body use="literal"/>

output>

<input>

<soap:body use="literal"/>

input>

<output>

<soap:body use="literal"/>

output>

This introduces the soap:body element from the SOAP language that defines the data type system to be used by SOAP processors, via the use attribute. The use attribute can be set to "encoding" or "literal".

13.3.8. The service, port, and endpoint elements

The service element simply provides a physical address at which the service can be accessed. It hosts the port element that contains this location information.

Example 13.15. The service and port elements establishing the physical service address.

<service name="EmployeeService">

<port binding="tns:EmployeeBinding" name="EmployeePort">

port>

service>

Because we are binding to the SOAP protocol, the port element contains a child soap:address element with the physical address information. Note that the port element is replaced with the endpoint element in version 2.0 of the WSDL specification.

13.3.9. The import element

WSDL definitions support a similar form of modularity as XSD schemas do. The import element can be used to import parts of the WSDL definition as well as XSD schemas.

Example 13.16. The import element referencing a schema document.

<import namespace="http://www.xmltc.com/tls/schemas/"

location="http://www.xmltc.com/tls/schemas/employee.xsd"/>

Note

See the Consider using modular WSDL documents guideline at the end of Chapter 15 for more information.

13.3.10. The documentation element

This optional element simply allows you to add descriptive, human-readable annotations within a WSDL definition. Developers can use this information when building service requestors or it can be programmatically retrieved through a service registry to aid the discovery of the service.

Example 13.17. The documentation element providing a description of the overall service interface.

<documentation>

Retrieves an XML document and converts it into the

native accounting document format.

documentation>

...

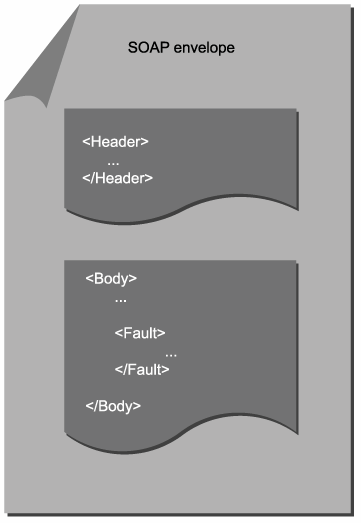
**SOAP language basics**

In the previous section we established that a WSDL definition intentionally separates abstract from concrete definition details. One of the benefits of doing so is that we can isolate communication protocols that implement the messaging required by a service from the implementation-neutral service interface. However, given that SOAP has become the messaging format of choice for SOA, we will very likely be binding our abstract interfaces to SOAP.

Within the service-oriented design process, we place a great deal of emphasis on hand crafting the WSDL definition, along with required XSD schema types. SOAP messages generally do not require as much hands on attention. We spend more time working with SOAP syntax in Chapter 17, where we explore WS-\* extensions that are implemented via SOAP headers. Still, this is as good of a time as any to introduce some basic parts of the SOAP language.

As we established in Chapter 5, the structure of SOAP messages is relatively simple. They consist of header, body, and fault sections, all encased in an envelope. Appropriately, the elements we describe in this section (Figure 13.5) are represented by the same names. (The manner in which SOA affects the utilization of SOAP is explored in the SOAP and SOA section in Chapter 14.)

Figure 13.6. The structure of a SOAP message document.



13.4.1. The Envelope element

The Envelope element represents the root of SOAP message structures. It contains a mandatory Body construct and an optional Header construct.

Example 13.18. The root Envelope construct hosting Header and Body constructs.

<Envelope xmlns ="http://schemas.xmlsoap.org/soap/envelope/">

...

...

Envelope>

13.4.2. The Header element

As explained in Chapter 5, the header portion of the SOAP message has become a key enabler of the feature set provided by WS-\* specifications. Most of these extensions are implemented on a message level and introduce new standardized SOAP header blocks destined to be embedded in the Header construct.

Example 13.19. The Header construct hosting a header block.

<Header>

0131858580-JDJ903KD

Header>

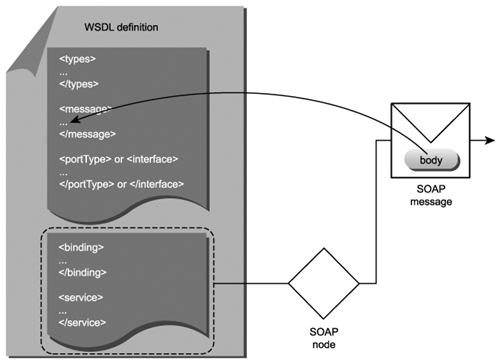
Header blocks also can be customized, as shown in Example 13.19, where the SOAP header is used to host a unique CorrelationID element. The mustUnderstand attribute indicates that the contents of the header must be understood by any receiver of the message that is required to process this header. If the mustUnderstand value is set to "0" the processing of this header becomes optional.

13.4.3. The Body element

This is the one required child element of the SOAP Envelope construct. It contains the message payload formatted as well-formed XML. The structure and naming used to define this part of the SOAP message relates to the style and use attributes discussed in the previous WSDL binding element description.

SOAP message Body constructs are defined within the WSDL message constructs which, as we've already established, reference XSD schema data type information from the WSDL types construct (Figure 13.7).

Figure 13.7. A SOAP message body defined within the WSDL message construct. The actual processing of the SOAP message via a wire protocol is governed by the constructs within the concrete definition.



Example 13.20. The contents of a sample Body construct.

<Body>

0131858580

Service-Oriented Architecture

Concepts, Technology, and Design

Body>

While SOAP header blocks can be processed actively during the transmission of a SOAP message, the SOAP body should not be touched. However, if allowed, intermediary services can still read and derive information from body content. For example, a correlation identifier used in a SOAP header can be generated based on the ISBN value shown in the preceding example.

13.4.4. The Fault element

The optional Fault construct provides a ready made error response that is added inside the Body construct. In the example that follows, this fault information is further sub-divided using additional child elements. The faultcode element contains one of a set of fault conditions predefined by the SOAP specification. Both the faultstring and detail elements provide human readable error messages, the latter of which can host an entire XML fragment containing further partitioned error details.

Example 13.21. The Fault construct residing within the Body construct.

<Fault> MustUnderstand header was not recognized The CorrelationID header was not processed by a recipient that was required to process it. Now a fault's been raised and it looks like this recipient is going to be a problem. Fault>

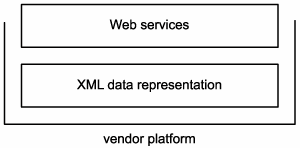
In our example a Fault construct is provided to respond to a MustUnderstand violation that may occur when a service expected to process the message correlation identifier fails to do so.

**Steps to composing SOA**

Regardless of the shape or size of your SOA, it will consist of a number of technology components that establish an environment in which your services will reside (Figure 14.1). The fundamental components that typically comprise an SOA include:

* an XML data representation architecture
* Web services built upon industry standards
* a platform capable of hosting and processing XML data and Web services

Figure 14.1. The most fundamental components of an SOA.



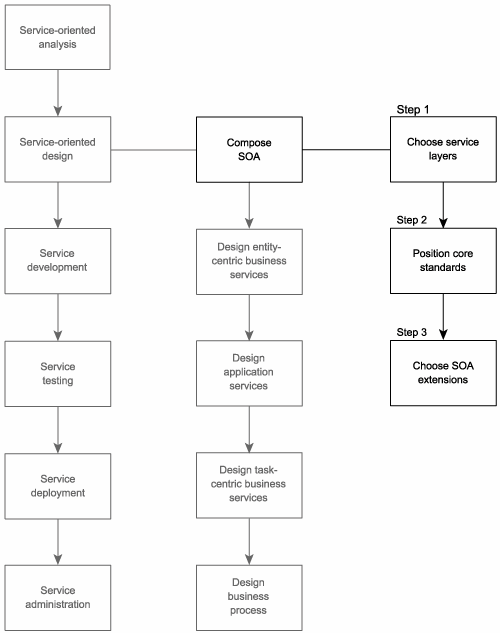
However, to support and realize the principles and characteristics we've explored as being associated with both the primitive and contemporary types of SOA requires some additional design effort.

Common questions that need to be answered at this stage include:

* What types of services should be built, and how should they be organized into service layers?
* How should first-generation standards be positioned to best support SOA?
* What features provided by available extensions are required by the SOA?

These issues lead to an exercise in composition, as we make choices that determine what technologies and architectural components are required and how these parts are best assembled.

Figure 14.2. Suggested steps for composing a preliminary SOA.



14.1.1. Step 1: Choose service layers

Composing an SOA requires that we first decide on a design configuration for the service layers that will comprise and **standardize logic representation** within our architecture. This step is completed by studying the candidate service layers produced during the service-oriented analysis phase and exploring service layers and service layer configuration scenarios.

14.1.2. Step 2: Position core standards

Next, we need to assess which core standards should comprise our SOA and how they should be implemented to best support the features and requirements of our service-oriented solution. The Considerations for positioning core SOA standards section provides an overview of how each of the core XML and Web services specifications commonly is affected by principles and characteristics unique to SOA.

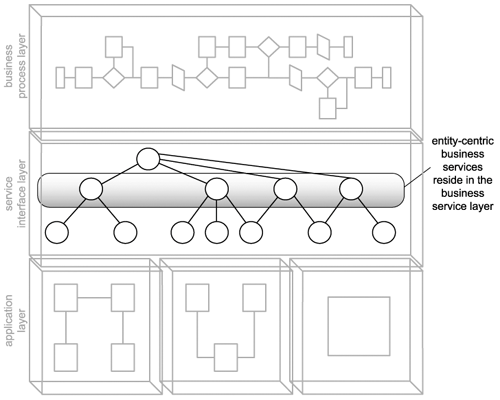
14.1.3. Step 3: Choose SOA extensions

This final part of our "pre-service design process" requires that we determine which contemporary SOA characteristics we want our service-oriented architecture to support. This will help us decide which of the available WS-\* specifications should become part of our service-oriented environment. The Considerations for choosing SOA extensions section provides some guidelines for making these determinations.

**Entity-centric business service design (a step-by-step process)**

Entity-centric business services represent the one service layer that is the least influenced by others. Its purpose is to accurately represent corresponding data entities defined within an organization's business models. These services are strictly solution- and business process-agnostic, built for reuse by any application that needs to access or manage information associated with a particular entity.

Figure 15.1. Entity-centric services establish the business service layer.

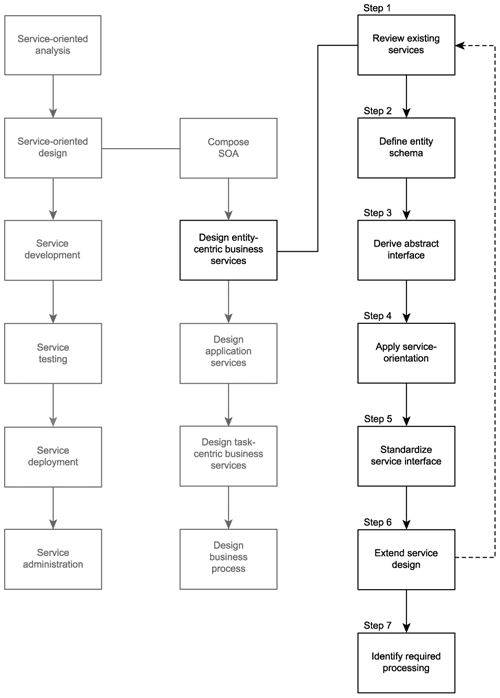


Because they exist rather atomically in relation to other service layers, it is beneficial to design entity-centric business services prior to others. This establishes an abstract service layer around which process and underlying application logic can be positioned.

15.2.1. Process description

Provided next is the step-by-step process description wherein we establish a recommended sequence of detailed steps for arriving at a quality entity-centric business service interface (Figure 15.2).

Figure 15.2. The entity-centric business service design process.



Note that the order in which these steps are provided is not set in stone. For example, you may prefer to define a preliminary service interface prior to establishing the actual data types used to represent message body content. Or perhaps you may find it more effective to perform a speculative analysis to identify possible extensions to the service before creating the first cut of the interface.

All of these can be legitimate approaches. The key is to ensure that in the end, design standards are applied equally to all service operations and that all processing requirements are accurately identified.

Let's begin now with the design of our entity-centric business service.

Step 1: Review existing services

Ideally, when creating entity-centric services, the modeling effort resulting in the service candidates will have taken any existing services into account. However, because service candidates tend to consist of operation candidates relevant to the business requirements that formed the basis of the service-oriented analysis, it is always worth verifying to ensure that some or all of the processing functionality represented by operation candidates does not already exist in other services.

Therefore, the first step in designing a new service is to confirm whether it is actually even necessary. If other services exist, they may already be providing some or all of the functionality identified in the operation candidatesorthey may have already established a suitable context in which these new operation candidates can be implemented (as new operations to the existing service).

Step 2: Define the message schema types

It is useful to begin a service interface design with a formal definition of the messages the service is required to process. To accomplish this we need to formalize the message structures that are defined within the WSDL types area.

SOAP messages carry payload data within the Body section of the SOAP envelope. This data needs to be organized and typed. For this we rely on XSD schemas. A standalone schema actually can be embedded in the types construct, wherein we can define each of the elements used to represent data within the SOAP body.

In the case of an entity-centric service, it is especially beneficial if the XSD schema used accurately represents the information associated with this service's entity. This "entity-centric schema" can become the basis for the service WSDL definition, as most service operations will be expected to receive or transmit the documents defined by this schema.

Note that there is not necessarily a one-to-one relationship between entity-centric services and the entities that comprise an entity model. You might recall in the service modeling example from Chapter 12, we combined Employee and EmployeeHistory entities into one Employee Service. In this case, you can either create two separate schemas or combine them into one. The latter option is recommended only if you are confident you will never want to split these entities up again.

Note

As demonstrated in the upcoming example, the WSDL definition can import schemas into the types area. This can be especially beneficial when working with standardized schemas that represent entities. (See the Consider using modular WSDL documents guideline for more information.)

Step 3: Derive an abstract service interface

Next, we analyze the proposed service operation candidate and follow these steps to define an initial service interface:

1. Confirm that each operation candidate is suitably generic and reusable by ensuring that the granularity of the logic encapsulated is appropriate. Study the data structures defined in Step 2 and establish a set of operation names.
2. Create the portType (or interface) area within the WSDL document and populate it with operation constructs that correspond to operation candidates.
3. Formalize the list of input and output values required to accommodate the processing of each operation's logic. This is accomplished by defining the appropriate message constructs that reference the XSD schema types within the child part elements.

Step 4: Apply principles of service-orientation

Here's where we revisit the four service-orientation principles we identified in Chapter 8 as being those not provided by the Web services technology set:

* service reusability
* service autonomy
* service statelessness
* service discoverability

Reusability and autonomy, the two principles we already covered in the service modeling process, are somewhat naturally part of the entity-centric design model in that the operations exposed by entity-centric business services are intended to be inherently generic and reusable (and because the use of the import statement is encouraged to reuse schemas and create modular WSDL definitions). Reusability is further promoted in Step 6, where we suggest that the design be extended to facilitate requirements beyond those identified as part of our service candidate.

Because entity-centric services often need to be composed by a parent service layer and because they rely on the application service layer to carry out their business logic, their immediate autonomy is generally well defined. Unless those services governed by an entity-centric controller have unusual processing requirements or impose dependencies in some manner, entity-centric services generally maintain their autonomy.

It is for similar reasons as those just mentioned that statelessness is also relatively manageable. Entity-centric services generally do not possess a great deal of workflow logic and for those cases in which multiple application or business services need to be invoked to carry out an operation, it is preferred that state management be deferred as much as possible (to, for example, document-style SOAP messages).

Discoverability is an important part of both the design of entity-centric services and their post-deployment utilization. As we mentioned in Step 1, we need to ensure that a service design does not implement logic already in existence. A discovery mechanism would make this determination much easier. Similarly, one measure we can take to make a service more discoverable to others is to supplement it with metadata details using the documentation element, as explained in the Document services with metadata guideline.

Step 5: Standardize and refine the service interface

Depending on your requirements, this can be a multi-faceted step involving a series of design tasks. Following is a list of recommended actions you can take to achieve a standardized and streamlined service design:

* Review existing design standards and guidelines and apply any that are appropriate. (Use the guidelines and proposed standards provided at the end of this chapter as a starting point.)
* In addition to achieving a standardized service interface design, this step also provides an opportunity for the service design to be revised in support of some of the contemporary SOA characteristics we identified in the Unsupported SOA characteristics section of Chapter 9.
* If your design requirements include WS-I Basic Profile conformance, then that can become a consideration at this stage. Although Basic Profile compliance requires that the entire WSDL be completed, what has been created so far can be verified.

Step 6: Extend the service design

The service modeling process tends to focus on evident business requirements. While promoting reuse always is encouraged, it often falls to the design process to ensure that a sufficient amount of reusable functionality will be built into each service. This is especially important for entity-centric business services, as a complete range of common operations typically is expected by their service requestors.

This step involves performing a speculative analysis as to what other types of features this service, within its predefined functional context, should offer.

There are two common ways to implement new functionality:

* add new operations
* add new parameters to existing operations

While the latter option may streamline service interfaces, it also can be counter-intuitive in that too many parameters associated with one operation may require that service requestors need to know too much about the service to effectively utilize it.

Adding operations is a straight-forward means of providing evident functions associated with the entity. The classic set of operations for an entity-centric service is:

* GetSomething
* UpdateSomething
* AddSomething
* DeleteSomething

Security requirements notwithstanding, establishing these standard operations builds a consistent level of interoperability into the business service layer, facilitating ad-hoc reusability and composition.

Note

Despite the naming suggestions listed here, when designing business services to reflect existing entity models, it is often beneficial to carry over the naming conventions already established (even if this means adjusting existing naming standards accordingly).

If entirely new tasks are defined, then they can be incorporated by new operations that follow the same design standards as the existing ones. If new functional requirements are identified that relate to existing operations, then a common method of extending these operations is to add input and output values. This allows an operation to receive and transmit a range of message combinations. Care must be taken, though, to not overly complicate operations for the sake of potential reusability. It often is advisable to subject any new proposed functionality to a separate analysis process.

Also, while it is desirable and recommended to produce entity-centric services that are completely self-sufficient at managing data associated with the corresponding entity domain, there is a key practical consideration that should be factored in. For every new operation you add, the means by which that operation completes its processing also needs to be designed and implemented. This boils down to the very probable requirement for additional or extended application services. As long as the overhead for every new operation is calculated and deemed acceptable, then this step is advisable.

Note that upon identifying new operations, Steps 1 through 5 need to be repeated to properly shape and standardize added extensions.

Step 7: Identify required processing

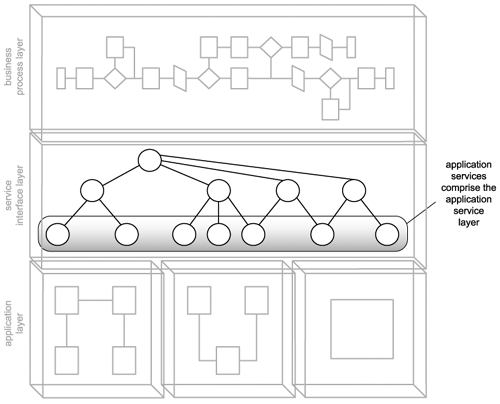
While the service modeling process from our service-oriented analysis may have identified some key application services, it may not have been possible to define them all.

Now that we have an actual design for this new business service, you can study the processing requirements of each of its operations more closely. In doing so, you should be able to determine if additional application services are required to carry out each piece of exposed functionality. If you do find the need for new application services, you will have to determine if they already exist, or if they need to be added to the list of services that will be delivered as part of this solution.

**Application service design (a step-by-step process)**

Application services are the workhorses of SOA. They represent the bottom sub-layer of the composed service layer (Figure 15.10), responsible for carrying out any of the processing demands dictated to them by the business and orchestration layers.

Figure 15.10. Application services establish the bottom sub-layer of the service layer.



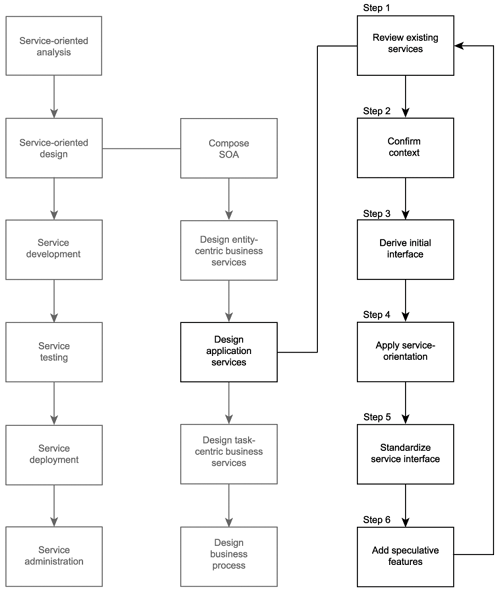
Unlike services in business-centric layers, the design of application services does not require business analysis expertise. The application service layer is a pure, service-oriented abstraction of an organization's technical environments, best defined by those who understand these environments the most.

Because of the many real-world and technology-specific considerations that need to be taken into account, application services can be the hardest type of service to design. Further, the context established by these services can be constantly challenged, as technology is upgraded or replaced, and as related application logic is built or altered.

15.3.1. Process description

Figure 15.11 provides a proposed service design process for creating application service interfaces. Note that all references made to "application services" in this and remaining chapters imply that they are reusable utility application services.

Figure 15.11. The application service design process.



When viewing Figure 15.11, you'll notice that this process shares a number of steps with the previous entity-centric business service process. This is because both application and entity-centric services establish reusable service logic and therefore rely on parent controllers to compose them into business process-specific tasks.

However, there are key aspects in how the two processes differ. Note, for example, how the confirmation of the operation grouping context is isolated into a separate step. Establishing context for application services is an important and much less clear-cut part of service design.

Further, there is no step in which processing requirements are defined. This is primarily due to the fact that application services are responsible for implementing the processing details required to carry out the business logic of their parent business services. This, of course, does not mean that processing requirements for application services do not exist. They do, only they are part of the design of the underlying service application logic. Because we are only designing a service interface at this stage, it is not considered part of the process.

Let's begin putting together the application service interface.

Step 1: Review existing services

More so with application services than with other types of reusable services, it is important to ensure that the functionality required, as per the application service candidate, does not, in some way, shape, or form, already exist. So it is very necessary to review your existing inventory of application services in search of anything resembling what you are about to design.

Additionally, because these services provide such generic functionality, it is worth, at this stage, investigating whether the features you require can be purchased or leased from third-party vendors. Because application services should be designed for maximum reusability, third-party Web services (which typically are built to be reusable) can make a great deal of sense, as long as required quality of service levels can be met.

Step 2: Confirm the context

When performing a service-oriented analysis it's natural to be focused on immediate business requirements. As a result, application service candidates produced by this phase will frequently not take the contexts established by existing application services into account.

Therefore, it is important that the operation candidate grouping proposed by service candidates be re-evaluated and compared with existing application service designs. Upon reassessing the service context, you may find that one or more operations actually belong in other application services.

Note

This step was not required as part of the entity-centric business service design, as the context of entity-centric services is predefined by the corresponding entity models.

Step 3: Derive an initial service interface

Analyze the proposed service operation candidates and follow the steps below to define the first cut of the service interface:

1. Using the application service candidate as your primary input, ensure that the granularity of the logic partitions represented by the operation candidates are appropriately generic and reusable.
2. Document the input and output values required for the processing of each operation candidate and define message structures using XSD schema constructs (which essentially establishes the WSDL types construct).
3. Complete the abstract service definition by adding the portType (or interface) area (along with its child operation constructs) and the necessary message constructs containing the part elements that reference the appropriate schema types.

Note that as generic units of processing logic, application services will be used by different types of business services. Each business service will be processing a different type of business document (invoice, purchase order, claim, etc.). Therefore, application services need to be designed in such a manner that they can process multiple document types. Depending on the nature of the information being processed, there are several design options.

Examples include:

* Create a set of operations that are generic but still document specific. For example, instead of a single Add operation, you could provide separate AddInvoice, AddPO, and AddClaim operations.
* Application services can be outfitted to support SOAP attachments, allowing a generic operation to issue a generic SOAP message containing a specific business document.

Step 4: Apply principles of service-orientation

This step highlights the four principles of service-orientation we listed in Chapter 8, as being those that are not intrinsically provided by the Web services platform (service reusability, service autonomy, service statelessness, and service discoverability).

Reuse was discussed in the service modeling process and is addressed directly in Step 5, where we look at making our application service as useful to potential service requestors as possible. However, the existing operation candidates also should be reviewed to ensure they are designed to be generic and reusable.

Autonomy is of primary concern when designing application services. We must ensure that the underlying application logic responsible for executing the service operations does not impose dependencies on the service, or itself have dependencies. This is where the information we gathered in Step 2 of the service-oriented analysis process provides us with a starting point to investigate the nature of the application logic each service operation needs to invoke. Step 6 provides an analysis that covers this and other technology-related issues.

Statelessness also may be more difficult to achieve with application services. Because they are required to interface with a variety of different application platforms, these services are subject to highly unpredictable implementation environments. Sooner or later, application services are bound to encounter challenges that impose unreasonable or inconsistent performance requirements (outdated legacy systems are known for this). Therefore, the best way to promote a stateless application service design is to carry out as much up-front analysis as possible. Knowing in advance what the performance demands will be will allow you to investigate alternatives before you commit to a particular design.

As with entity-centric services, discoverability can be an important part of evolving the application services layer. To guarantee that this design does not overlap with the logic already provided by other application services, a discoverability mechanism is useful. This becomes more of an infrastructure requirement that can be planned as part of an SOA implementation. However, the Document services with metadata guideline still applies, as application services should be supplemented with as much metadata as possible.

Step 5: Standardize and refine the service interface

Even though the role and purpose of application services differs from other types of services, it is important that they be designed in the same fundamental manner. We accomplish this by ensuring that the resulting application service WSDL definition is based on the same standards and conventions used by others.

Following is a list of recommended actions you can take to achieve a standardized and streamlined service design:

* Apply any existing design standards relevant to the service interface. (For a list of suggested standards, review the guidelines provided at the end of this chapter.)
* Review any of the contemporary SOA characteristics you've chosen to have your services support and assess whether it is possible to build support for this characteristic into this service design.
* Optionally incorporate WS-I Basic Profile rules and best practices to whatever extent possible.

Step 6: Outfit the service candidate with speculative features

If you are interested in delivering highly reusable application services, you can take this opportunity to add features to this service design. These new features can affect existing operations or can result in the addition of new operations. For application services, speculative extensions revolve around the type of processing that falls within the service context.

Of course, before actually adding speculative extensions to the application service, you should repeat Step 1 to confirm that no part of these new operations already exists within other services. Additionally, when adding new extensions, Steps 2 through 5 also need to be repeated to ensure that they are properly standardized and designed in alignment with the portion of the service interface we've created so far.

Step 7: Identify technical constraints

At this point we have created an ideal service interface in a bit of a vacuum. Unlike our business services, application services need to take low-level, real-world considerations into account.

We now need to study and document the processing demands of each service operation more closely. First, for each operation, write a list of the processing functions required for the operation to carry out its processing. Then, for every entry on this list, find out exactly how the processing of the function will need to be executed in the existing technical environment.

The types of details we are specifically looking for are:

* The physical connection point of the particular function. (In other words, what components need to be invoked, what API functions need to be called, or which adapters need to be activated.)
* Security constraints related to any part of the processing.
* Response time of each processing function.
* Availability of the underlying system performing the processing function.
* Environmental factors relating to service deployment location.
* Technical limitations of underlying application logic (especially when exposing legacy systems).
* Administration requirements imposed by the service.
* Potential SLA requirements.

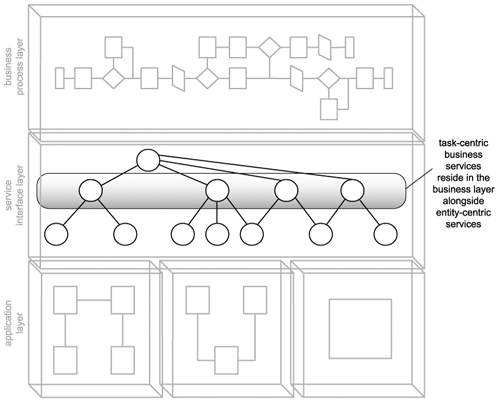
After characteristics of individual processing functions have been gathered, they need to be viewed collectively. For example, individual response times need to be added to calculate the overall estimated execution time of the operation. The result of this study is typically a series of constraints and limitations imposed by the technical environment onto our service interface. In some cases, the restrictions will be so severe that an operation may need to be significantly augmented.

Note that when transitioning an organization toward an enterprise-wide SOA, there is a tendency to want to service-orient everything. However, it is important to identify which processing requirements cannot be fulfilled by the Web services technology set. It may not make sense to expose some portions of underlying legacy application logic as Web services. Either way, it is worth reminding ourselves that even though this book focuses on the creation of services as Web services, SOA is in fact an implementation-neutral architectural model and service-orientation is an implementation-neutral design paradigm. Existing forms of application logic not made available through Web services can still be modeled as services. This is of particular relevance to application services, where exposing application logic through a Web service may not always be the right decision. For example, façade components are often created to encapsulate functionality from different sources and to then expose a distinct context representing a set of reusable functions. This results in a legitimate service, which may, in the future, still be expressed via a Web service.

**Task-centric business service design (a step-by-step process)**

The process for designing task-centric services usually require less effort than the previous two design processes, simply because reuse is generally not a primary consideration. Therefore, only the service operation candidates identified as part of the service modeling process are addressed here.

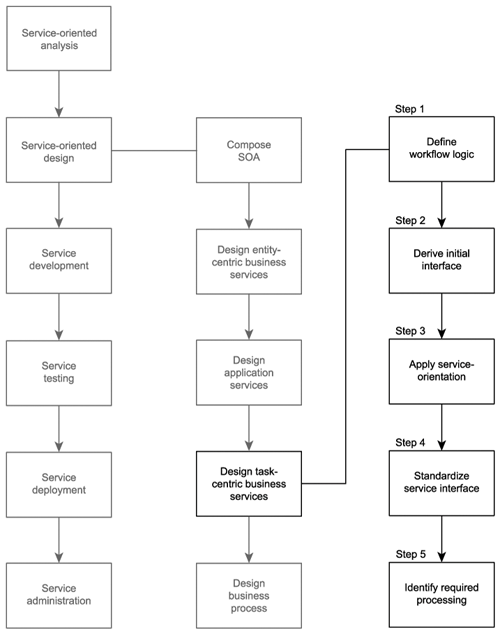
Figure 15.15. Task-centric business services can comprise the business service layer, along with entity-centric neighbors.



15.4.1. Process description

As shown in Figure 15.16, this process starts off with a new kind of step in which workflow logic is mapped out. This is because task-centric business services are expected to contain and govern portions of business processes.

Figure 15.16. The task-centric business service design process.



Note that there is no step encouraging you to extend the service design beyond the feature set you already defined in the service modeling stage. As previously mentioned, providing a generic and reusable interface is not a priority for task-centric services.

Time now to begin our service design.

 Step 1: Define the workflow logic

Task-centric services typically will contain embedded workflow logic used to coordinate an underlying service composition. Our first step, therefore, is to define this logic for every possible interaction scenario we can imagine. If you performed the mapping exercise in the Identify candidate service compositions step of the service modeling process in Chapter 12, then you will have preliminary composition details already documented.

Because we are designing our task-centric business service after our entity-centric and application service designs have been completed, we will need to revisit these scenario documents and turn them into concrete service interaction models.

Different traditional modeling approaches can be used to accomplish this step (we use simple activity diagrams in our case study examples). The purpose of this exercise is to document each possible execution path, including all exception conditions. The resulting diagrams also will be useful input for subsequent test cases.

Note

The workflow logic does not reside in the service interface we are designing in this process. We are defining workflow logic for the purpose of extracting the message exchanges with which this service will be involved. This provides us with information that helps us define types, operations, and message formats.

Step 2: Derive the service interface

Follow these suggested steps to assemble an initial service interface:

|  |  |
| --- | --- |
| **1.** | Use the application service operation candidates to derive a set of corresponding operations. |
| **2.** | Unlike previous design processes, the source from which we derive our service interface this time also includes the activity diagrams and the workflow logic we documented in Step 1. This information gives us a good idea as to what additional operations our task-centric service may require. |
| **3.** | Document the input and output values required for the processing of each operation and populate the types section with XSD schema types required to process the operations. |
| **4.** | Build the WSDL definition by creating the portType (or interface) area, inserting the identified operation constructs. Then add the necessary message constructs containing the part elements that reference the appropriate schema types. |

Step 3: Apply principles of service-orientation

Before we get too far ahead in our service design, it is beneficial to take another look at the four service-orientation principles we covered in Chapter 8, which are not automatically provided to us through the use of Web services (service reusability, service autonomy, service statelessness, and service discoverability).

Reuse opportunities for task-centric services are much more rare than for entity-centric and application services. This is because task-centric services represent a portion of workflow logic specific to a business process. However, reuse still can be achieved. The Take into account the potential cross-process reusability of the logic being encapsulated and Consider the potential intra-process reusability of the logic being encapsulated modeling guidelines in Chapter 12 address this and still are applicable to this process.

Because they almost always act as parent controller services in compositions, the autonomy of task-centric services is generally dependent on the autonomy of the underlying child services. A consistent state of autonomy can therefore be challenging to maintain.

Task-centric services contain workflow logic that may impose processing dependencies in service compositions. This can lead to the need for state management. However, the use of document-style SOAP messages allows the service to delegate the persistence of some or all of this state information to the message itself.

It is always useful for services to be discoverable, but the need for task-centric services to be discovered is not as pressing as with other, more generically reusable services. Regardless, task-centric services can achieve reuse, and their existence should be known to others. Therefore, the Document services with metadata guideline provided at the end of this chapter also is recommended.

Step 4: Standardize and refine the service interface

Although task-centric business services will tend to have more creative operation names, existing conventions still need to be applied. Here is the standard list of recommended actions you can take to achieve a standardized and streamlined service design:

* Incorporate existing design standards and guidelines. (A set of recommended guidelines is provided at the end of this chapter.)
* Ensure that any chosen contemporary SOA characteristics are fully supported by the service interface design.
* Take WS-I Basic Profile standards and best practices into account.

With regard to design standards relating to operation granularity, some leniency may be required to accommodate the processing of the service's embedded workflow sequence logic. Also, task-centric business services can benefit from reusing existing WSDL modules, in particular, XSD schema definitions.

Step 5: Identify required processing

To carry out their share of a solution's process logic, task-centric services can compose application and both entity-centric and additional task-centric business services. Therefore, the implementation of a task-centric service interface requires that any needed underlying service layers are in place to support the processing requirements of its operations.

Because this is the last of our three service design processes, all required supporting aplication services need to be identified. They may consist of services that already existed and/or services we just designed during the previous application service design process. The design of process logic within the task-centric business service may also reveal the need for additional application services that haven't yet been considered.