THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

WS-Addressing • WS-Policy
WSDL • XML Schema
SOAP • Namespaces
Web Services
WS-* • SOA

"The New Bible for WS- Design and SOA."*
Sascha Kuhlmann
Global Program
Lead Enterprise
Architecture
SAP

Web Service Contract Design & Versioning for SOA

Foreword by David Chappell

Thomas Erl, Anish Karmarkar, Priscilla Walmsley, Hugo Haas, Umit Yalcinalp, Canyang Kevin Liu, David Orchard, Andre Tost, James Pasley

PRENTICE HALL

Library of Congress Cataloging-in-Publication Data:

Web service contract design and versioning for SOA / Thomas Erl ... [et al.].

p. cm.

ISBN 0-13-613517-X (hbk. : alk. paper) 1. Web services. 2. Computer network architectures. 3. Computer architecture. I. Erl, Thomas. II. Title: Web service contract design and versioning for Service Oriented Architecture.

TK5105.88813.E76 2008

006.7'8—dc22

2008025188

ISBN-13: 978-0-13-613517-3
ISBN-10: 0-13-613517-X
Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing September 2008

Editor-in-Chief
Mark Taub

Managing Editor
Kristy Hart

Copy Editor
Language Logistics

Development Editors
Dmitry Kirsanov
Songlin Qiu

Indexer
Cheryl Lenser

Proofreader
Williams Woods Publishing

Composition
Jake McFarland
Bumpy Design

Graphics
Zuzana Cappova
Tami Young

Photos
Thomas Erl
Johan Odendaal

Foreword

I have often commented that learning the Web services set of specifications (WS-*) by reading them is like trying to understand the English language by reading Webster's dictionary. This is because every concept, every word is expressed in terms of other words that have yet to be understood, resulting ultimately in cyclical definitions. How could one effectively learn a verbal language that way?

Nearly ten years in the making, the WS-* specifications are the result of an industry-wide effort to define interoperable interfaces, transports, and metadata, using XML and XML Schema as the means for doing so. The specifications are intentionally broken up into small atomic units so that individual applications or infrastructure offerings from commercial vendors and open source may implement them in order to ease adoption and move us all closer to platform-independent interoperability. They are also composable with each other in that they may be used together. But how?

In all fairness, the WS-* specifications are self-explanatory enough to get a rough understanding of the purpose of each one if you read them long and hard enough. Through arduous scrutiny of jargon and angle brackets, eventually there is enough that sinks in to reach a critical mass of basic understanding.

Grasping their purpose and using the WS-* specifications as a reference for implementation is only the first level of knowledge. In fact most programmers and architects by now are familiar with the common trio (WSDL, XML Schema, and SOAP). This does not, however, imply that the use of these specifications is intuitively obvious. And what about the other specs in the stack? At last count, there were nearly 50. Can they all be learned together? How does one sort through them all and figure out which ones are the

most important to pay attention to and which will have the most impact on their organization if used properly? What is the impact if used improperly? Even if the details are partially hidden behind an SOA platform, there are still architectural issues to be considered.

Enter this book. It is about technologies and practices related to designing and governing Web service contracts in support of SOA and service-orientation. Written by experts who have themselves contributed to the creation and development of the WS-* specifications, this book provides guidance to demonstrate how the most relevant Web service contract technologies work together as a framework—insight that all too often gets left behind on the cutting room floor after countless conference calls and face-to-face meetings among working groups, technical committees, task forces, and sub-committees. The book further helps the reader to understand how the WS-* technologies can be best leveraged and evolved in support of SOA and service-orientation.

This book provides the reader with common understanding across a broad range of WS-* technologies as they pertain to service contracts, service versioning, policy management, and SOA governance in order to enable the reader to leverage the full capabilities that are part of modern WS-* platforms. Issues such as the pros and cons and proper use of the many standardized interoperable message exchange patterns are explored, as well as how to take full advantage of WS-Addressing. In addition, this book provides in-depth discussion on governance issues, particularly as it pertains to service contract design, service versioning, and service policy. Insight and guidance is provided on when and how to design WSDL operations that are intelligently combined with XML Schema and WS-Policy statements to produce service contracts that are conducive to effective versioning.

We hope you enjoy this body of work. Your SOA projects are likely already under way or are about to begin. You are building new architectures for your organization that are going to be in place for decades to come. Many years of hard work have gone into the WS-* set of specifications, and this book will help you to reap the benefits of that work to create truly loosely-coupled and flexible service contracts that will mitigate the impact of change on your organization.

—*David A. Chappell, Technologist and author of several books and papers on interoperability, SOA, and Enterprise Service Bus*

Preface

After we completed this manuscript, I checked the schedule and noticed our original start date. From the initial kick-off call during which everyone was given the green light to begin writing their chapters to the day I had to hand over the manuscript to Prentice Hall for indexing spanned a period of about 32 months. I initially didn't think too much of it because I already knew this project had taken over two years. But when I looked at that number again sometime later, it struck me.

The time it has taken for this book to be developed and authored is actually comparable to the time it originally took for several of the XML and Web services-based technology specifications covered in this book to be developed into fully ratified standards.

Though a curious statistic, this comparison doesn't do the subject matter justice. The development processes these technology standards were subject to are on entirely different levels, in that they were vastly complex both from human and technology perspectives.

There's the human element that emerges in the technical committee that is tasked with the responsibility of producing a standard. Such a committee will be comprised of members with different agendas, different perceptions, and different personalities. So many differences can turn a standards development process into a rollercoaster of group dynamics, ranging from strong teamwork to stages of scrutiny, confrontation, and even raw tension. Trying to achieve a consensus in an active technical committee is not for the weak at heart.

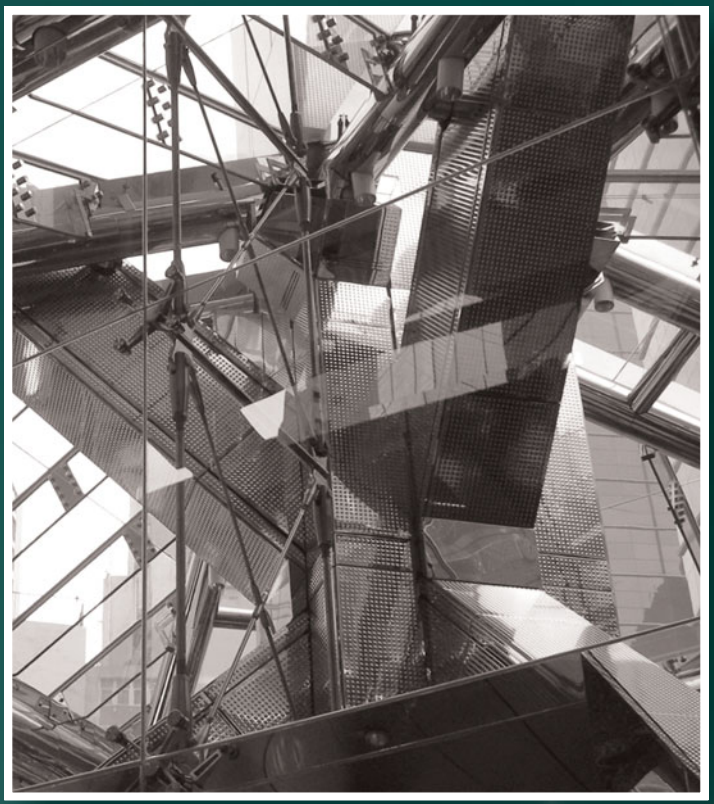
And then there's the technology element, which is reflected in the deliverables produced by the committee. Technical specifications are meticulously crafted and worded and revised and reworded in continuous, patient, and sometimes mind-numbingly tedious

cycles. But despite best efforts, creating a new language or vocabulary that will meet the ever-escalating needs and expectations of the industry as a whole is a daunting prospect. Not to mention that there is a constant possibility that the particular standard a committee might have spent a good part of their lives working on will be overshadowed by a competing effort or perhaps even rejected by the industry altogether.

But amidst these challenges, there have been many success stories. In a way, this book is a testament of this in that it documents a collection of respected and widely-recognized de facto standards that have established themselves as important IT milestones.

Ultimately, though, this book is about you, the reader. It was written for you to fully leverage what these technology standards have to offer. As successful as these technologies have been, what counts in the end is how effective they are for you.

—Thomas Erl



Chapter 4

Anatomy of a Web Service Contract

Web service contracts can range in content, depth, and complexity. To fully appreciate the intricacies and design options of how Web service contracts can be structured, we first need to decompose this structure in order to understand its individual parts and the mechanics that make these parts work together.

The purpose of this chapter is to explain the Web service contract from a conceptual and structural perspective without yet getting into the details of how the contract is actually developed through markup code.

We start exploring contract structure by breaking the contract down into a set of primary parts. This allows us to describe what these parts are, what technologies can be used to create them, and how they can relate to each other. Subsequent chapters will then drill down into each of the mentioned parts and technologies.

4.1 What is a Web Service Contract?

A Web service contract is essentially a collection of metadata that describes various aspects of an underlying software program, including:

- the purpose and function of its operations
- the messages that need to be exchanged in order to engage the operations
- data models used to define the structure of the messages (and associated validation rules used to ensure the integrity of data passed to and from the messages)
- a set of conditions under which the operations are provided
- information about how and where the service can be accessed

Several different and cooperating technologies are required to formally define this metadata, as explained later in the *Technologies Used to Create Web Service Contracts* section.

Fundamental Structure

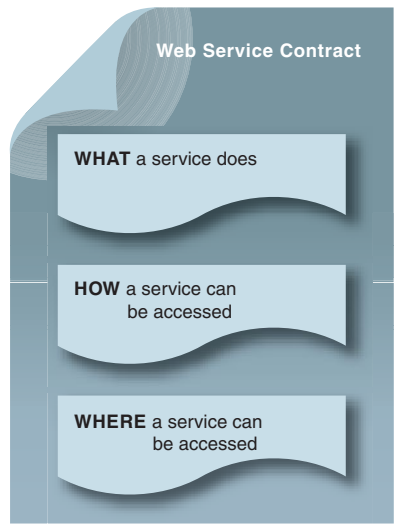
Web service contracts are organized into a basic structure that reflects a relatively clear separation of "what," "how," and "where" as follows:

- **What** is the purpose of the service and its capabilities?
- **How** can the service be accessed?
- **Where** can the service be accessed?

When potential consumer program designers evaluate a service, they need to know *what* the service is capable of doing and under what conditions it can carry out its capabilities. If what's offered is what consumer designers need, then they must be able to determine *how* and *where* to access the service.

As illustrated in Figure 4.1, the service contract is organized into sections that individually address these three questions.

Figure 4.1
A Web service contract defines what a service offers and how and where it can be accessed.



In addition to providing flexibility as to how a service can be located and consumed, this clean separation allows different parts of the contract to be owned and developed at different stages of the service delivery lifecycle by different members of a project team.

For example, architects and analysts can focus solely on the "what" part of a service when it is being conceptualized and designed. The "how" and "where" parts don't usually become relevant until developers actually build and deploy the contract as part of the overall service implementation.

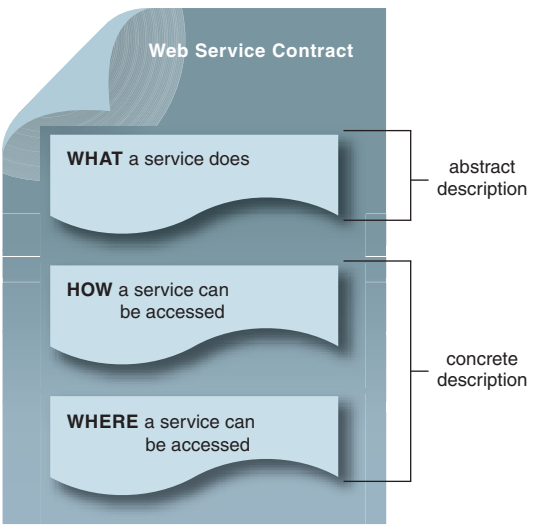
Abstract and Concrete Descriptions

There are formal terms used to represent the three fundamental parts we just covered. As shown in Figure 4.2, the "what" portion of the Web service contract is referred to as the *abstract description*. This part is essentially responsible for expressing the contract's public interface (or API).

The balance of the Web service contract is represented by the *concrete description*, providing the implementation and communication details necessary for consumer programs to locate and use the service at runtime.

The concrete description encompasses the previously described "how" and "where" parts. An example of a characteristic that would be considered a "how" part is the wire format and protocol necessary to exchange messages with the service.

Figure 4.2
An abstract description establishes the technical interface independent of implementation details. A concrete description adds deployment-specific details about how and where to access a service.



NOTE

The terms "abstract description" and "concrete description" originated with the W3C, the standards organization responsible for developing most of the technologies covered in this book. From an implementation perspective, the abstract description is also very much concrete in that it ends up forming the actual physical interface that service consumer programs end up connecting to.

SUMMARY OF KEY POINTS

- The term "abstract description" is commonly used to refer to the "what" part of the contract, whereas the term "concrete description" represents both the "how" and "where" parts.
- Abstract and concrete descriptions are frequently created during different stages of the service delivery lifecycle by different members of the project team.

4.2 The Parts of a Web Service Contract

Having just established a high-level structure of a Web service contract, let's drill down a bit to explore how abstract and concrete descriptions are further sub-divided into smaller parts.

NOTE

The following sections cover only the primary parts of the Web service contract. There are many other, more specialized definitions that will be introduced and explored in subsequent chapters.

Primary Parts of the Abstract Description

Figure 4.3 provides us with a *logical* view of the parts that comprise the abstract description. This means that when this section of a contract is actually created, these parts will be physically structured differently (as shown later in the *A Physical View of the Abstract Description* section).

The following sections are numbered to correspond with the numbers on Figure 4.3.

1. Port Type (Interface) Definition

The definition of the *port type* is the cornerstone of a Web service contract. Think of a port as a place of entry (such as a port used to receive ships for transferring cargo). A Web service contract can have one or more "types" of ports. Each port type essentially represents a technical interface or access point.

Note that for reasons we'll explain later, it is sometimes more appropriate to refer to the port type definition as the "interface definition."

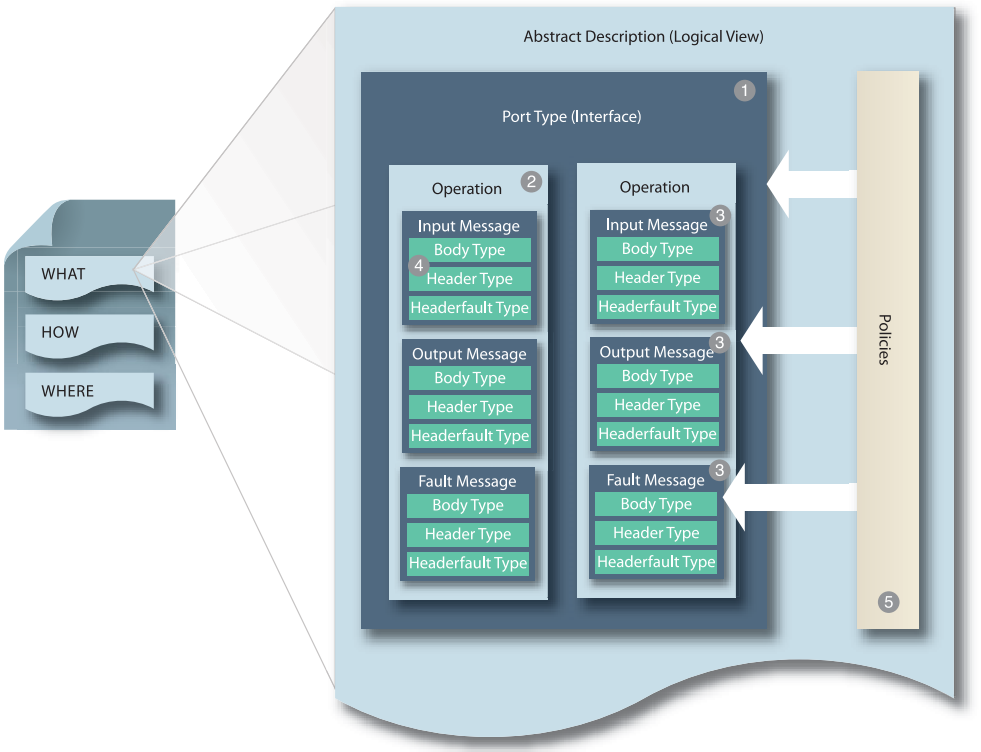


Figure 4.3
A logical view of the primary parts of an abstract description.

2. Operation Definition

For a Web service to be used by another program, it needs to expose externally consumable capabilities or functions. These are referred to as operations and are expressed as *operation definitions*.

A port type acts as a container for a set of related operations.

3. Message Definition

When a Web service is being invoked via one of its operations, the consumer program needs to exchange data with it in order for the Web service to perform the requested function. A *message definition* essentially establishes a "mini-structure" for the data to be transmitted.

There are three types of message definitions:

- *Input* – a message sent to the Web service by a consumer program
- *Output* – a message sent by the Web service to the consumer program
- *Fault* – an error notification message sent by the Web service to the consumer program

An operation can contain one or all of these message definitions but would generally not contain only an output or a fault message.

4. Type Definition

The data for a given input, output, or fault message needs to have a pre-defined structure of its own. That way, the Web service knows whether or not it is receiving or sending valid data. This data structure is established in a *body type definition*.

Messages further support the inclusion of message-specific metadata (information about the message that accompanies the message). For this form of supplementary data, a *header type definition* is also provided.

Finally, just as message transmissions can encounter error conditions, so can the processing of message metadata. Therefore, an additional *headerfault type definition* is available for when a response to a metadata-related error condition needs to be sent.

NOTE

The use of the headerfault type definition is not common in practice.

5. Policy Definition

All of the previous definitions in the abstract description have collectively defined the functional aspects of the Web service contract. *Policy definitions* can be created to extend the technical interface by expressing further behavioral requirements and characteristics of the service.

A Web service contract can have any number of supplementary policies. The horizontal arrows in Figures 4.3 and 4.4 indicate that polices can be specific to certain parts of the abstract description.

Note that polices can also be applied to parts of the concrete description.

A Physical View of the Abstract Description

So far we've been focusing on the logical view of the abstract description because it is conceptually important to understand it from this perspective. However, this is a book about technology, and the hundreds of pages that follow this chapter are all concerned with how Web service contracts are physically built.

Therefore, it's time that we start learning about how the organization of the various definitions we've described so far differs in a real-world contract document. As shown in Figure 4.4, the purpose behind the physical structure is to promote reuse in support of flexible and normalized Web service contracts.

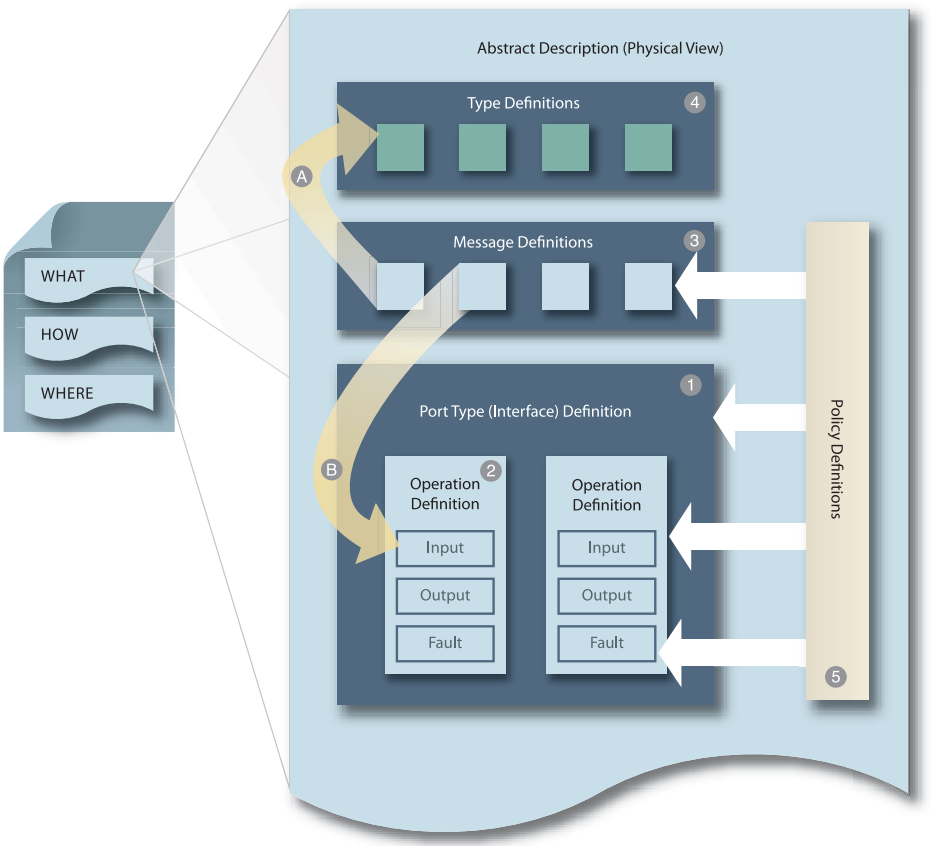


Figure 4.4
A physical view of the primary parts of an abstract description.

By following the numbers in Figure 4.4 we can trace back the actual location of the previously described definitions. While the port type definition (1) and policy definitions (5) are pretty much in the same place, we can see that the message definitions (3) and the type definitions (4) are no longer part of the port type or operation definitions (2).

Type and message definitions are positioned in separate parts of the contract so that they can be reused as follows:

A. Each type definition can be used by multiple message definitions.

B. Each message definition can be used by multiple operations (as input, output, and fault messages).

Although Figure 4.4 shows a series of available types, it does not make a distinction between a body, header, or headerfault type. All three types can exist as individual type definitions (4) but are then bundled together into one message definition (3). The body portion of this message is then associated with an input, output, or fault message as part of the operation definition (2).

NOTE

The header and headerfault parts are assigned to messages in the concrete description, the reason being that the messaging protocol to which the abstract description will be bound may or may not support their use.

WSDL 2.0 Changes to Physical View

We've intentionally avoided mentioning specific technologies so far in this chapter in order to establish as much of an abstract perspective of Web service contracts as possible. But it is worth noting that this book covers two different versions of the WSDL language used to express the Web service contract structure.

WSDL is explained later in this chapter. At this stage, we just need to point out that the physical view we just described complies with WSDL version 1.1. In version 2.0, the message definitions section (3) is removed, and operation definitions (2) refer directly to individual type definitions (4).

NOTE

In the upcoming sections dedicated to the parts of the concrete description, there is no distinction between a logical and physical view.

Primary Parts of the Concrete Description (Part I)

A concrete description supplements the abstract description with implementation-specific details required for the Web service to be invoked and communicated with by service consumer programs.

As we established earlier, the concrete description corresponds to the "how" and "where" sections of a Web service contract. We'll focus on these sections individually, starting with the "how" considerations addressed by the binding-related definitions illustrated in Figure 4.5.

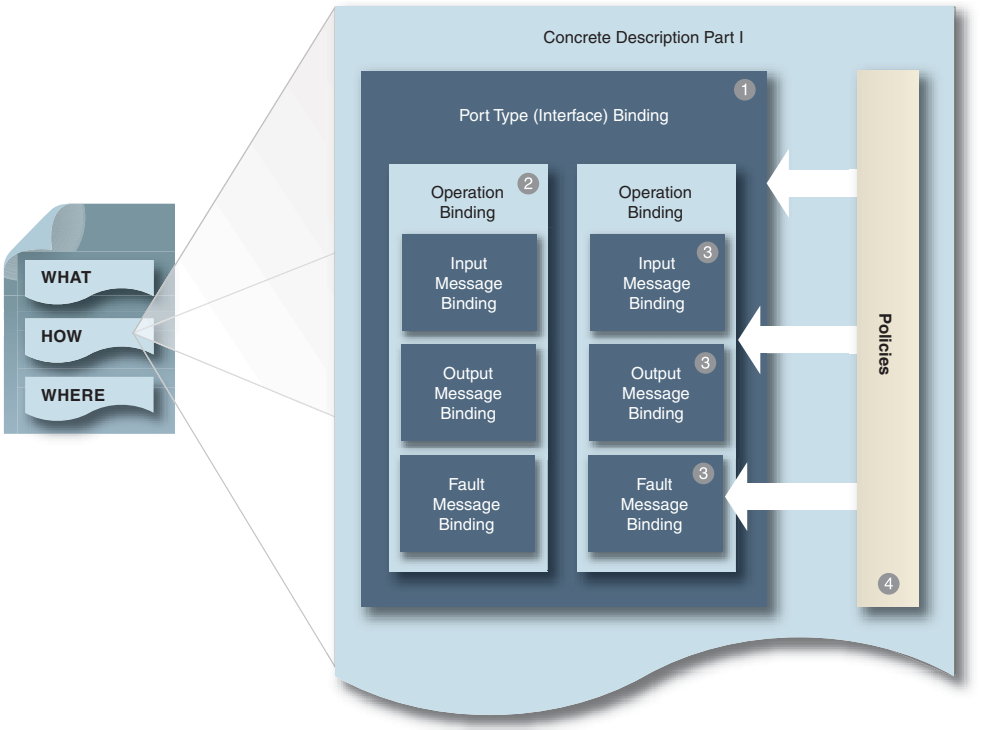


Figure 4.5
The primary parts of the binding portion of the concrete description. (Note that this represents both logical and physical views.)

The following sections are numbered to correspond with the numbers on Figure 4.5.

1. Port Type (Interface) Binding Definition

A *binding definition* details the communication technology that can be used by consumer programs to invoke and interact with the Web service.

A port type definition within an abstract description can be associated with one or more binding definitions, each of which contains the following two pieces of information:

- *Messaging Protocol Binding* – Defines a technology or industry standard (the message protocol) that specifies the format in which a message should be packaged by the sender and then unpackaged by the receiver.
- *Transport Protocol Binding* – Defines the communications technology (the transport protocol) with which the message should be transmitted across a network.

2. Operation Binding Definition

An *operating binding definition* represents the same type of binding definition as the port type binding, except that it is specific to the operation only. If the operation does not have this binding specified, it inherits the binding settings from its parent port type.

3. Message Binding Definitions

As with the operation binding, the *message binding definition* also represents a granular form of binding definition that, in this case, is specific to an input or output (or fault) message. If a message-specific binding is not provided, the message inherits the binding settings from its parent operation or its parent port type (if an operation binding is not present).

It is also here in the message binding definition that the header and headerfault parts of a message get associated with an input, output, or fault message that was defined in the abstract description. The reason again for making this association here is because not all message protocols support the use of header and headerfault parts.

4. Policy Definition

As with the definitions of an abstract description, policies can also be defined for the individual parts of a concrete description. Policies for binding definitions tend to be related to the configuration and runtime requirements of a particular messaging or transport protocol.

Primary Parts of the Concrete Description (Part II)

This final portion of the Web service contract structure focuses on the "where" section, as shown in Figure 4.6.

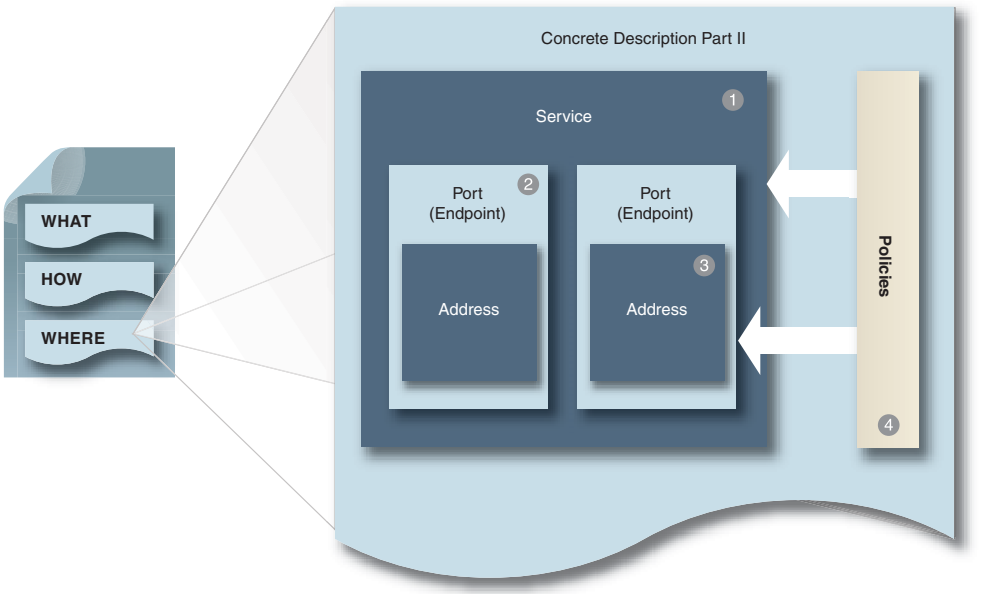


Figure 4.6
The primary parts of the service portion of the concrete description. (Note that this represents both logical and physical views.)

The following sections are numbered to correspond with the numbers on Figure 4.6.

1. Service Definition

The *service definition* simply groups related port (endpoint) definitions together. The port definition is explained shortly.

2. Port (Endpoint) Definition

Each binding definition needs to be associated with a port definition (also referred to as an endpoint definition), which simply acts as a container for the address definition (explained shortly). The same port definition can be used by different port type, operation, or message binding definitions.

3. Address Definition

An *address definition* establishes a physical network address. Depending on the communication technology (transport protocol) defined by the port type, operation, or message binding definition that is referencing the parent port definition for this address definition, the actual address value might be a Web URL, an email address, or some other type of transport-specific address.

4. Policy Definition

Policies can be created for address-related definitions in the concrete description. This may be necessary when certain requirements or characteristics specific to the network address need to be expressed.

How Parts of a Contract Relate to Each Other

As you may have already gathered, Web service contracts are highly modular. Let's take a minute to describe some of the relationships these parts can have:

- On a high level, one abstract description can have one or more associated concrete descriptions, each specifying a different messaging and/or transport protocol.
- One port type (interface) definition can have multiple operation definitions.
- Each operation definition can reference multiple message definitions.
- Each message definition can reference multiple type definitions.
- One port type (interface) definition can be associated with multiple binding definitions.
- Each binding definition can have multiple port (endpoint) definitions.
- A service definition can group multiple port (endpoint) definitions.
- Policies can be applied to most of these definitions.

Figure 4.7 illustrates these relationships by providing an example of one port type definition that is associated with two port type binding definitions (relationship A), each with its own port definition (relationship B).

Based on the numbers used in this diagram, we can further assume that the port type definition (1), the operation definition (2), and a message definition (3) each has its own corresponding binding definition. Also, the port type binding definition is associated with a port (4) that establishes its address.

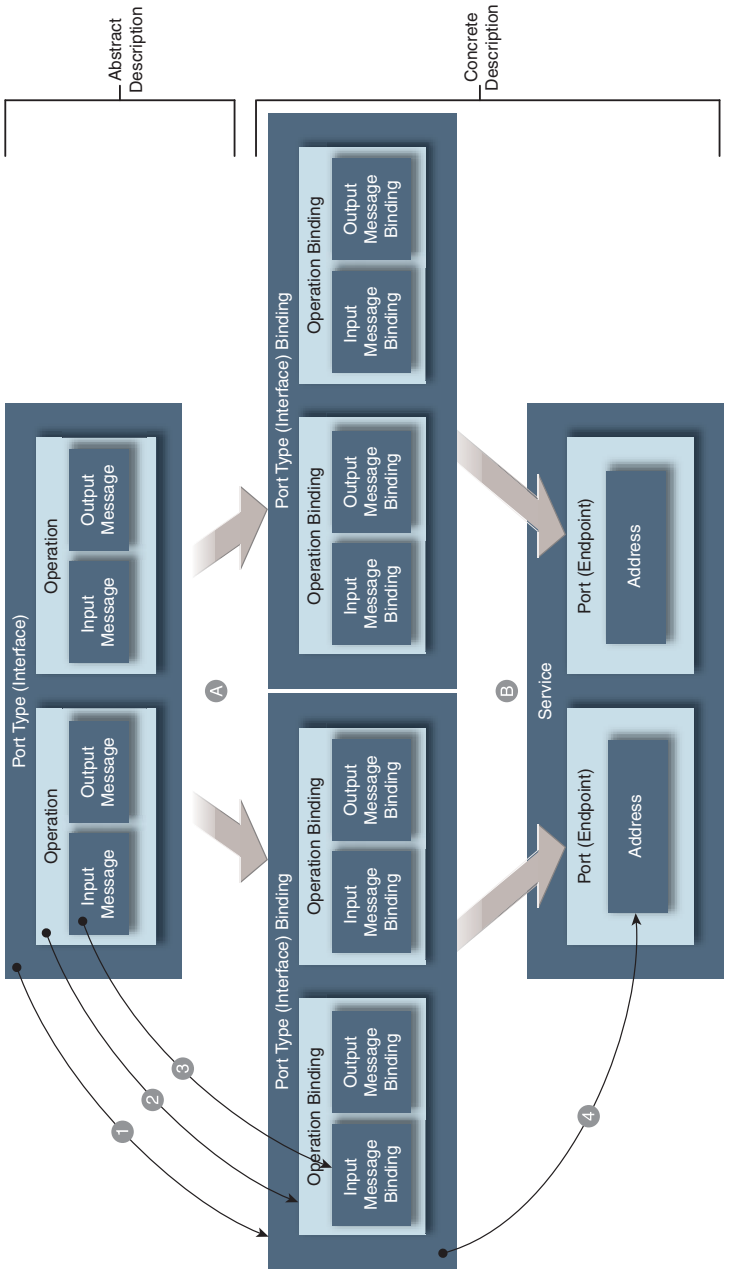


Figure 4.7

An example of how abstract and concrete descriptions can relate to each other. (Note that fault messages have been omitted from this diagram.)

These relationships are discussed in detail in subsequent chapters. At this point it's just helpful for us to understand that various types of relationships can exist.

The Primary Parts of a Message

It's important to remember that building Web service contracts is more than just creating an interface to underlying service logic. The contract defines, in a very specific manner, the actual interaction requirements of the Web service. And, as you may have gathered from the previous two sections, both abstract and concrete descriptions express these interaction requirements by defining input and output messages and the technologies used to support the processing of these messages.

Let's therefore take the time to familiarize ourselves with the basic parts of a message. As shown in Figure 4.8, messages are wrapped in envelopes that have a basic structure of their own.

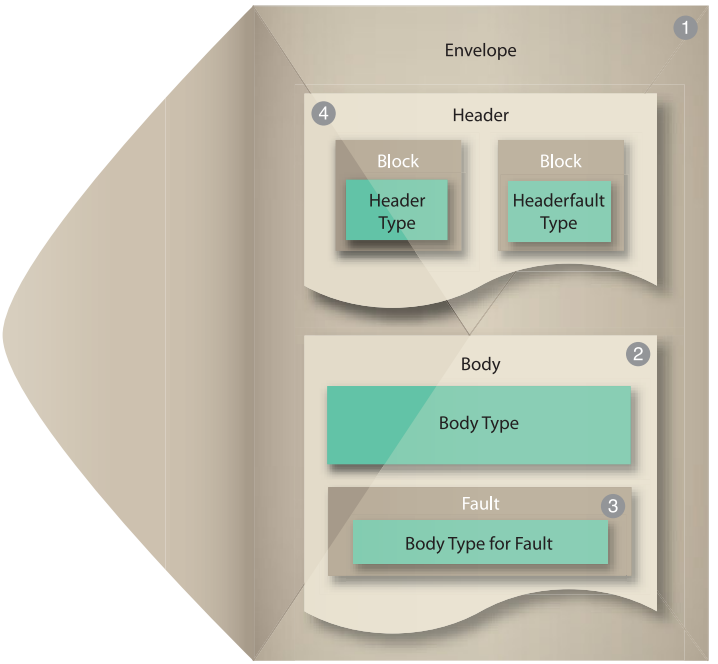


Figure 4.8
The fundamental structure of a message envelope.

Note that the following parts are not referred to as "definitions" because they represent an actual manifestation of what was established in the Web service contract definitions. In other words, runtime processors use the Web service contract definitions to generate these parts of the message (at runtime).

The following sections are numbered to correspond with the numbers on Figure 4.8.

1. Envelope

The scope of a message is defined by its *envelope*. This name is appropriate because it describes the message as a container of information. When discussing Web services, the terms "envelope" and "message" are frequently used interchangeably.

2. Body

The *body* represents a part of the message reserved for the data or the document being transported. Therefore, the type of information that usually resides within the message body is persistent in that its lifespan is greater than the time it takes for the message to be transmitted.

You'll notice the body part containing the *body type*. This relates back to the body type definition established in the abstract description. That type definition determined the structure for this portion of the message envelope.

Because a message exists as an actual body of data that is being transmitted somewhere, the body type in the message envelope would typically be populated with real data (like an invoice document) organized as per the body type definition from the message definition of the abstract description.

3. Fault

A message can be solely intended as a notification from the Web service to a consumer program that an error has occurred. The details of the error are housed in the fault part of the message envelope, which resides within the body section.

The fault part also has a type associated with it. This type corresponds to the fault type definition that was part of the abstract description's message definition.

4. Header

The *header* part of a message provides a location in which supplementary data about the message (metadata) can be stored. This data is usually temporary or non-persistent in nature in that its lifespan is generally equivalent to or shorter than the duration of the message transmission. A header allows metadata to be organized into sub-divided sections called *header blocks*.

Headers can contain many different types of header blocks, each dedicated to providing a distinct piece of supplementary data. This information is used by different programs to either help route a message or process its contents.

The first header block in Figure 4.8 is structured as per the header type definition that was established in the message definition of the abstract description. Similarly, the second header block is based on the headerfault type definition from the abstract description.

NOTE

Of all these parts, only the message envelope and the body are required. The header, header blocks, and fault parts are optional. Typically, only a body type or fault type is present within a given message (not both). Similarly, a message will either contain header types or headerfault types.As previously mentioned, whether the fault part is included in a message is based on which message definitions are present in the abstract description for a given operation.

With regards to header and headerfault parts, these do not always need to be defined as part of the Web service contract. In fact, in practice it has become more common for headers to be added, modified, and removed during the runtime processing of the message. This is explained in detail in Chapter 15.

SUMMARY OF KEY POINTS

- The primary parts of an abstract description are the port type, operation, message, type, and policy definitions.
- The primary parts of the concrete description are the port type binding, operation binding, message binding, service, port, and address definitions, as well as associated policy definitions.
- Definitions from abstract and concrete descriptions can have a variety of relationships.
- Abstract and concrete descriptions jointly define how real-life input and output messages are created.

4.3 Technologies Used to Create Web Service Contracts

Now that we've described the primary parts of a Web service contract and also how they can relate to each other, let's identify the development technologies suitable for creating these parts.

Each of these technologies shares the following characteristics:

- *Language* – Every part of a Web service contract is defined by writing code that conforms to a markup language. This is different from a traditional programming language, in that we do not compile our final results into a software program. We simply use the languages to create definitions that exist as textual documents; easy to access and read, but also reliant upon supporting platform programs for run-time processing.
- *XML-Based* – The markup grammar used to write these languages is expressed in XML. Therefore, you end up writing a Web service contract as a set of XML documents for the purpose of exchanging messages that also exist as XML documents and themselves carry data that is further represented with XML.
- *Vendor-Neutral* – Web service contract technologies were all developed by the same industry standards organization (the W3C) through rigorous, multi-year processes. Each was owned by a technical committee comprised of members from different vendor and practitioner organizations, and the final design of most of these language features required a consensus across this committee. Therefore, each of these technologies is also considered a legitimate industry standard.

More information regarding standards development processes, procedures, and stages is provided in Appendix B. Also note that these next sections only briefly introduce each technology; more in-depth coverage is provided in subsequent chapters.

"Element" vs. "Construct" vs. "Definition"

Before we introduce the technologies, let's first establish some additional terminology. If you are familiar with XML, you know that all of these XML-based languages are comprised of vocabularies of pre-defined elements, similar to those used in HTML (another Web language that was ratified into a standard through the W3C).

An additional term you'll see used in these next sections and throughout the remaining chapters is *construct* or *element construct*. A "construct" is a term used to refer to an element that contains nested child elements (in which case the element is implemented with an opening and closing tag). Think of a construct as an "element container."

For example, an HTML table exists as a table construct comprised of various nested elements (and nested constructs).

There are no rules as to when an element is referred to as a construct. It is just helpful at times to communicate that a particular part of a contract is represented as a container of elements.

The individual parts and definitions of a Web service contract that we've been explaining all exist as element constructs that contain further details. We make a distinction between a definition and a construct (or element) when we need to discuss how a part of a contract is implemented (coded) as opposed to designed.

So a definition, a construct, and an element can all refer to the same thing, just from different perspectives.

As we get into the details of these technology languages, we'll use a style convention whereby specific language element names are displayed in a different font. For example, we might state that the port type definition is created using the `portType` construct which is comprised of opening and closing `portType` tags.

Web Services Description Language (WSDL)

WSDL is considered to be the most fundamental and important technology for Web services development because it has become the de facto language for writing Web service contracts. Using WSDL, contract structure is defined and then further detailed and extended with the other technologies covered in this section.

As you may have gathered, we've already been discussing WSDL in this chapter. The basic Web service contract structure illustrated in previous diagrams has represented a WSDL definition, and the abstract and concrete descriptions are the fundamental parts of a WSDL structure (Figure 4.9).

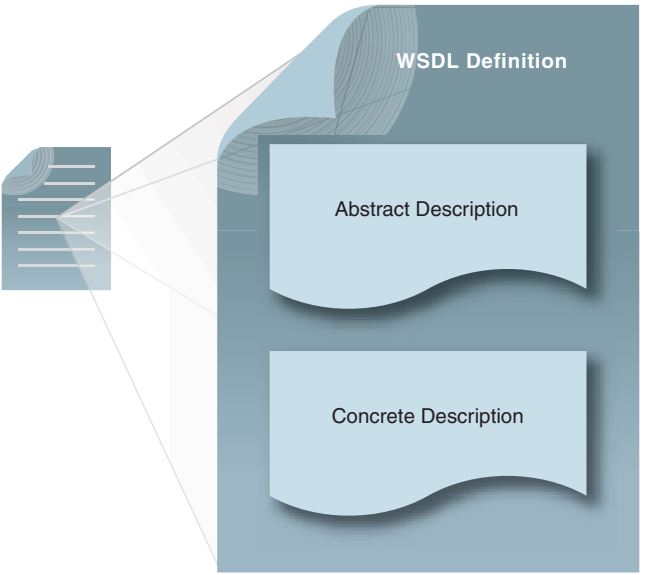


Figure 4.9
The basic WSDL definition structure that we have been exploring throughout this chapter.

A WSDL document ends up acting as somewhat of a container in which elements from almost all other languages covered in this book converge. Chapters 7 and 8 introduce the WSDL language as it applies to abstract and concrete descriptions respectively. As you explore how these parts of the WSDL definition are assembled, you will be able to see just how much code in the WSDL document does *not* originate from the WSDL language.

Chapter 9 provides further WSDL coverage specific to the WSDL 2.0 standard, and Chapters 14 and 15 drill down into a wide range of advanced topics.

XML Schema Definition Language (XML Schema)

XML Schema provides a formal language for defining the structure and validation constraints of XML documents. Because XML is a native part of the WSDL language and because Web services operate by exchanging XML messages, XML Schema is a highly complementary technology for creating Web service contracts.

Schemas serve as the part of a contract that describes the detailed structure of messages. They define what elements and attributes may appear in the message, in what order, and what type of data they are allowed to contain.

All of the type definitions displayed in the previous figures are assumed to be created using the XML Schema language. The types section displayed earlier in the physical

view of an abstract description (Figure 4.4) is technically part of the WSDL language, but its primary purpose is to house XML Schema elements.

XML Schema code can be embedded directly within WSDL documents or kept as separate documents and then referenced from the WSDL definitions. Each approach has its pros and cons.

When XML schemas exist as independent files, you can establish a many-to-many relationship between the schema documents and the WSDL message definitions. One schema might be used to describe several messages, particularly if they are related and share types. Similarly, each message definition can reference several different XML Schema types originating from different XML Schema documents.

NOTE

In case you're wondering why sometimes the word "schema" is capitalized and other times it isn't, this is explained in the "XML Schema" vs. "XML schema" section at the beginning of Chapter 6.

Most often, you will be designing your own schema from the ground up, based on your specific requirements. At other times, you will be using an industry schema that was developed by an industry organization, a vendor, or even a private corporation. When using an industry standard XML schema, you may wish to reuse only parts of it or extend it to meet your specific needs.

There are multiple ways to structure XML documents, just like there are a number of ways to design a database. Once you have decided on an XML structure (i.e. which elements and attributes you want to use) there is also more than one way to express that structure in an XML Schema definition.

Figure 4.10 provides a preview of common XML Schema constructs that we will be covering in Chapter 6. You will have the freedom to create a variety of different types based on XML Schema features that provide a series of built-in types and allow for the assembly of composite types commonly used to express complex structures and even entire business document structures.

Introductory XML Schema coverage is provided in the next chapter. Chapters 12 and 13 then delve into more advanced topics related to complex message design.

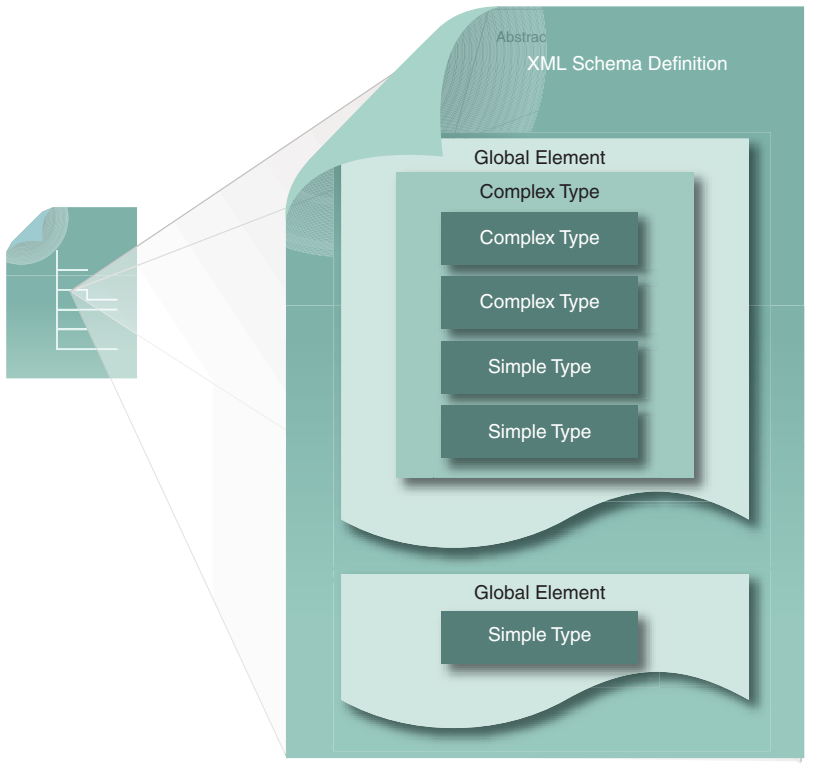


Figure 4.10
A logical view of a sample XML Schema definition structure comprised of different types that can then be used to define the structure of WSDL message definitions. Note the symbol on the left used to represent an XML Schema document.

WS-Policy Language

The WS-Policy standard specifies a policy framework specifically for Web services that enables behavior-related constraints and characteristics to be expressed as *machine-readable* metadata. This means that we can use this language to extend the base Web service contract provided by WSDL and XML Schema in order to add requirements or promote characteristics of the Web service that supplement the definitions in the abstract and concrete descriptions.

Figure 4.11 shows us that policy definition documents are just as modular as WSDL and XML Schema documents. As with XML Schema types, individual policy requirements (also referred to as policy assertions) can be combined or related to each other in order to form more complex composite policies (called policy expressions). You even have the ability to offer consumers a choice between two or more policies via the use of policy alternatives.

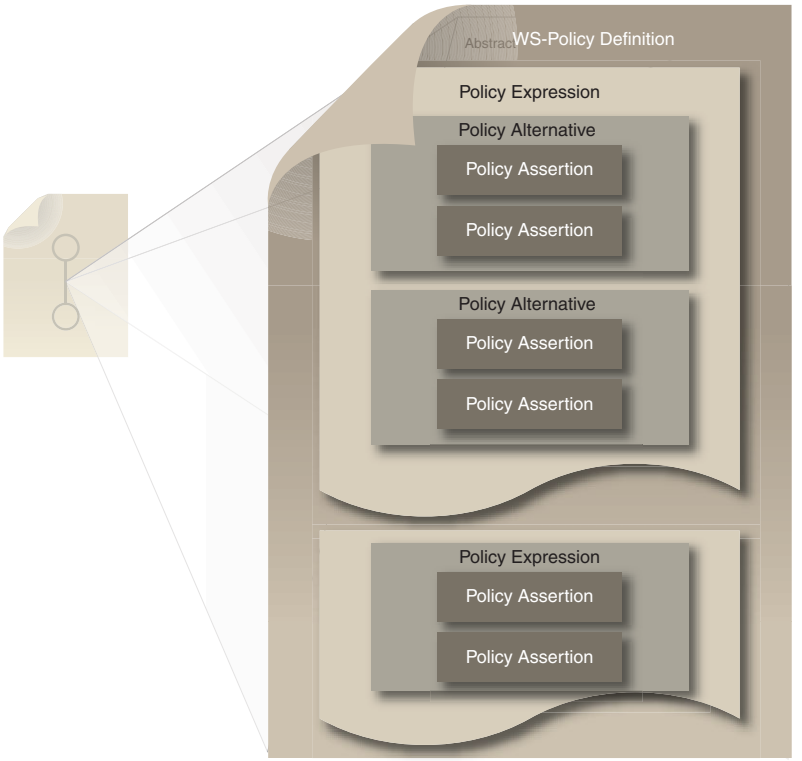


Figure 4.11
A sample WS-Policy definition with two composite policy expressions, each comprised of multiple policy assertions. Note the symbol on the left used to represent a policy document.

Another similarity that WS-Policy shares with XML Schema is how it relates to WSDL. Policies can also reside within or outside of the WSDL document. The related WS-PolicyAttachment language is used to associate (attach) policies to the various definitions in the abstract and concrete descriptions.

The WS-Policy and WS-PolicyAttachment languages are introduced in Chapter 10. More advanced topics are covered later in Chapters 16 and 17. Part of this coverage includes an exploration of how policy assertions are used by other languages to implement their features.

SOAP Language

SOAP provides a messaging format and mechanism that supports the standardized exchange of XML messages between software programs. SOAP has become the most prevalent messaging protocol used in conjunction with WSDL.

The SOAP language is used to express the structure of XML-based envelopes that host message definitions populated with actual data. Although at runtime SOAP language elements are usually auto-generated, SOAP is very much a technology related to the design-time creation of Web service contracts.

SOAP messages and message exchanges are fundamentally defined within the message and operation definitions of the WSDL document, and various SOAP language elements are further added to the concrete description in order to bind them to SOAP as a messaging protocol.

SOAP messages are most commonly:

- transmitted via HTTP
- described using XML Schema types
- associated with WSDL definitions in the concrete description

As shown in Figure 4.12, a standard SOAP message is divided into a set of parts. These parts are individually defined by XML Schema types via WSDL message definitions.

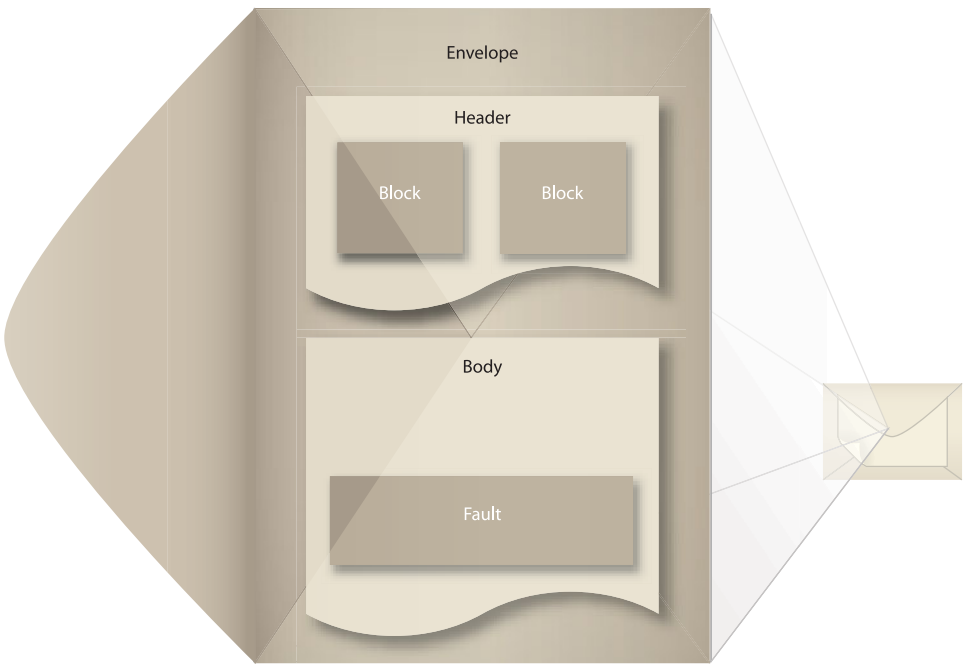


Figure 4.12
The SOAP message structure is defined by the SOAP standard and expressed with the SOAP language. Note the use of the symbol on the right to indicate a SOAP message in this book.

There are many supplemental technologies that extend SOAP messages via the use of SOAP header blocks. Many of these technologies are part of the WS-* specifications (also known as the second-generation Web services platform).

SOAP is introduced in Chapter 11. Advanced message design considerations are covered in Chapters 18 and 19 in relation to the WS-Addressing standard which provides fundamental routing and Web service instance identification header blocks.

Technologies for the Abstract Description

Figure 4.13 shows a high level mapping between the primary abstract description definitions and the related Web service technologies that can be used to create these definitions.

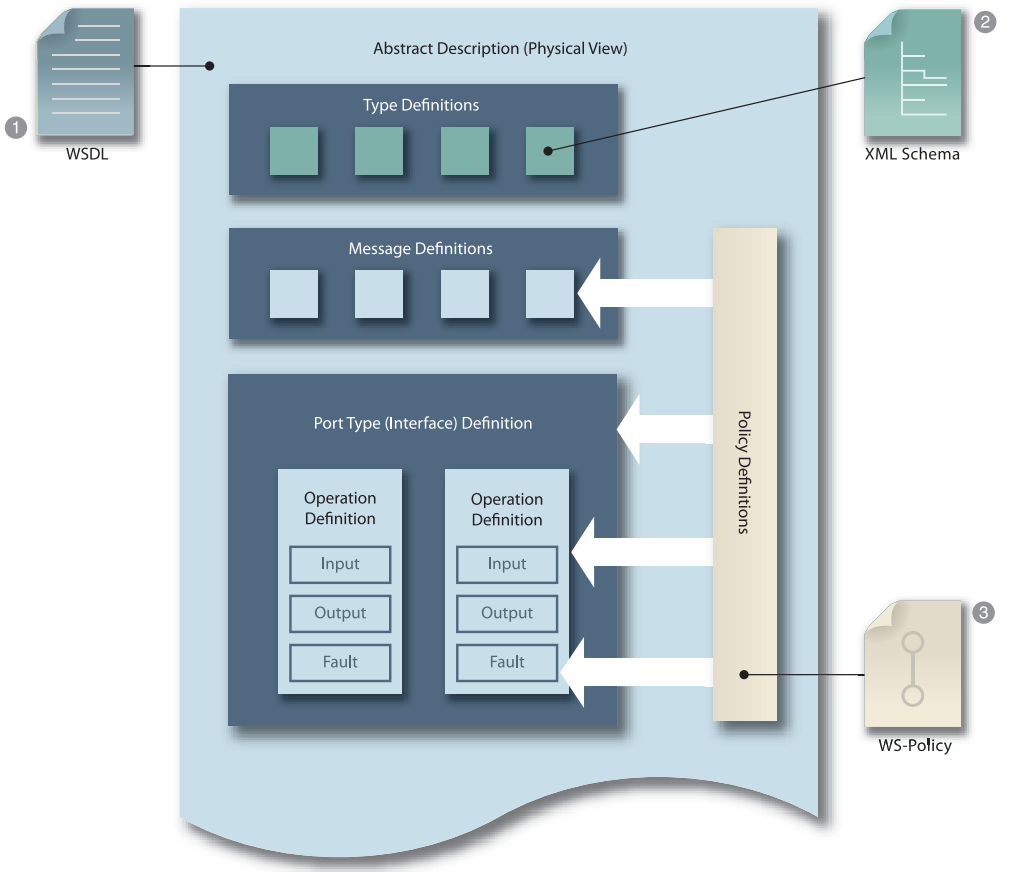


Figure 4.13
A high level mapping of the abstract description parts and related industry technologies.

What this figure tells us is that the overall structure of the abstract description (including the port type, operation, and message definitions) is created using WSDL (1).

The types part of a Web service contract is created collectively with WSDL and XML Schema (2) code in that the WSDL language provides a container construct wherein XML Schema code can be placed to express the various XML Schema types.

Finally, policy definitions can be separately created using the WS-Policy language (3). These policies can then be attached to the different WSDL definitions using the related WS-PolicyAttachment language.

Technologies for Concrete Descriptions

Let's now turn our attention to the technologies used to build a concrete description for a Web service. Figure 4.14 shows us the primary concrete description parts as they relate to three of the technology languages we just introduced.

In this depiction of a concrete description, we again establish how the WSDL (1) language is responsible for creating the overall structure, including the port type, operation and message binding definitions; as well as the service, port, and address definitions.

All of the constructs used to create these definitions can be further supplemented with special SOAP language statements (2) that allow the definitions to be directly associated with the SOAP messaging protocol.

And, as with the abstract description, all parts can be further supplemented with policy statements expressed using the WS-Policy language (3) as well as the related WS-PolicyAttachment language.

The WS-I Basic Profile

The Web Services Interoperability Organization (WS-I) develops specifications dedicated to establishing baseline interoperability between Web services using the technologies we just described.

The primary WS-I specification we'll be making reference to throughout this book is the Basic Profile, a document that essentially consists of a series of requirements and recommendations as to how Web services-related technology languages should be used together. Wherever appropriate, we'll be highlighting WS-I Basic Profile guidelines to supplement and provide further insight into contract design-related content.

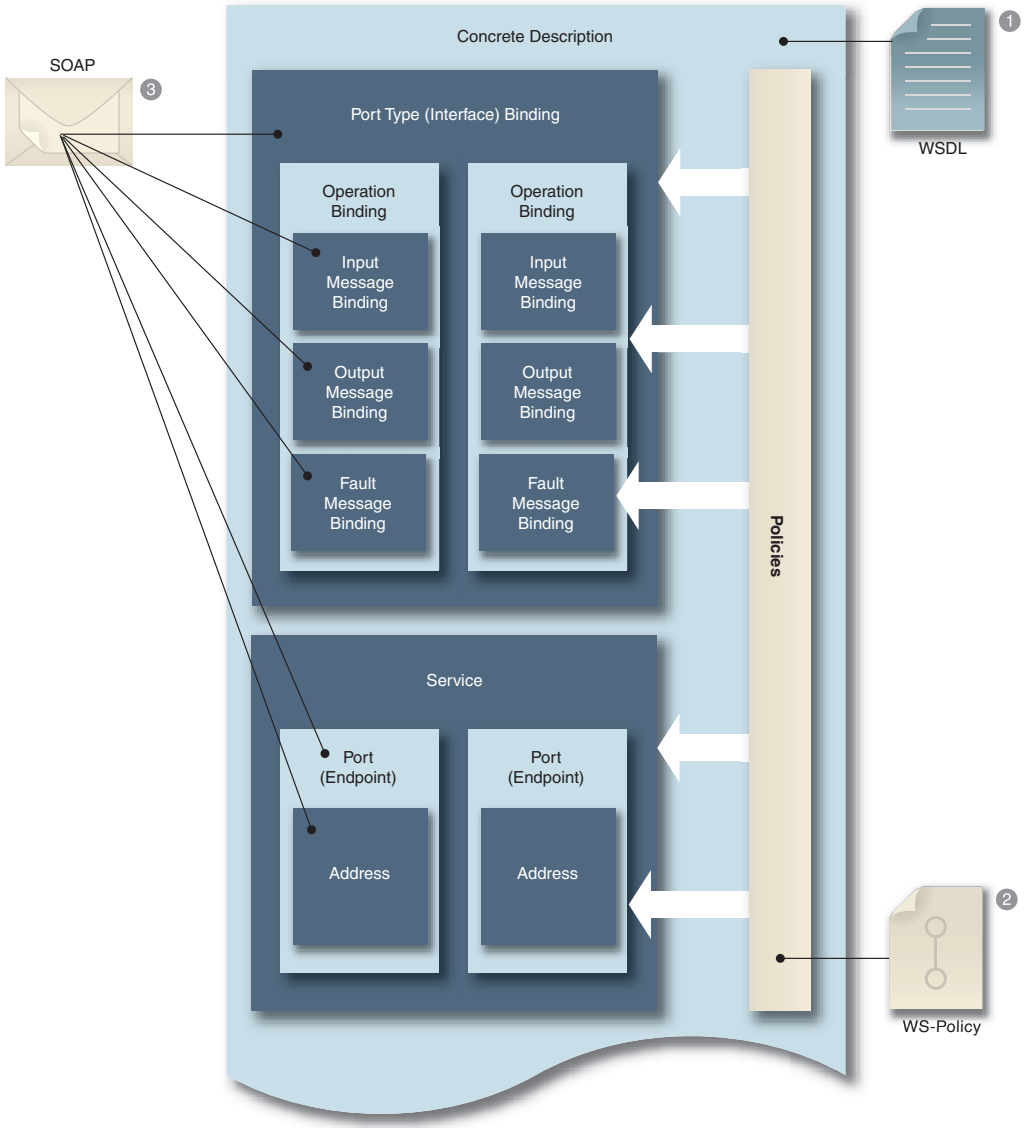


Figure 4.14
A look at the concrete description parts and the technology languages used to create them.

NOTE

In addition to the WS-I Profile, which is light on XML Schema guidelines, the W3C XML Schema Patterns for Databinding Working Group has documented a set of best practices for designing schemas for Web services. The XML Schema patterns are divided into two categories, basic and advanced, and there is a separate specification for each:

- Basic patterns use only the basic features of XML Schema and should already be correctly implemented by all toolkits.
- Advanced patterns are those that may still cause some tool interoperability problems today, but should be considered recommended patterns for the future.

To view the actual specifications for any of the languages covered in this book, visit www.soaspecs.com.

SUMMARY OF KEY POINTS

- The primary technology languages used to create the abstract description are WSDL, XML Schema, and WS-Policy.
- The primary technology languages used to create the concrete description are WSDL, SOAP, and WS-Policy.
- A WSDL document will be comprised of code originating from all of these languages.
- XML Schema and WS-Policy definitions can optionally be placed into standalone documents that are then referenced from within the WSDL definition.

4.4 Guidelines for Using Web Service Contract Technologies

It is not difficult to learn the mechanics of building Web service contracts. Once you get to know the technologies we've been discussing in this chapter, you will be able to author custom contracts with relative ease.

As we already established, every primary technology we cover in this book exists as an XML-based, industry-standard, markup language. Even though each serves a different purpose, all share a common syntax and many common conventions.

While learning these languages is a fundamental requirement to customizing your Web service contracts, it is not the most important skill, nor is it the most challenging part of creating effective contracts

Writing a Web service contract and *designing* an effective Web service contract for a given service as part of a federated service inventory are two different things. This is where technology features, design philosophies, and business requirements converge. This book primarily attempts to help you with the first two items.

Keep in mind that these technology languages are merely tools. You can just as easily use them to build a bad Web service contract as you could a good one. It is therefore helpful to remember that the decisions you make when designing the different parts of a contract will eventually impact the usability, flexibility, and governance burden of your services.

Before we proceed to the series of chapters that explore these technologies in detail, here are a few guidelines to keep in mind.

Auto-Generation Tools

Many development tools allow you to generate XML code for Web service contract languages. While this is useful for learning purposes, it may not be suitable for designing standardized Web service contracts in support of service-orientation.

The Standardized Service Contract design principle requires that service contracts be designed in accordance with existing design standards. The Service Loose Coupling principle further explains that contracts generated from underlying parts of a service's implementation can inherit negative forms of coupling that will make the service burdensome to own and govern.

These considerations also tie directly into the Canonical Schema and Canonical Expression design patterns that promote the use of design standards within the boundary of a service inventory.

Flexibility vs. Restrictiveness

One common dilemma in XML-based design, and indeed in software engineering in general, is the tension between flexibility and restrictiveness. At one end of the spectrum, a contract could be highly flexible in that it might allow just about any consumer to send it any type of content. The advantage of this approach is that you will never have to change the service contract when the message structure changes.

However, this technique also has its weaknesses. If you don't define a message structure, you can't pre-define validation logic in the contract layer, and you have no formal documentation of what your service can do. Applications that process the messages will have no idea of what to expect.

On the other hand, you could write a very rigid schema for your contract that must be modified every time there is the slightest change or addition to the message definition. Such contracts are brittle and have a ripple effect on the solutions that consume and form dependencies on the service. You may want this kind of rigidity if you are implementing a service that processes messages requiring strict precision (such as bank transactions), but such rigidity often does more harm than good.

With service-orientation there is the tendency to be less restrictive with technical contracts than with traditional object and component interfaces. The priorities advocated by the Service Abstraction principle and the Validation Abstraction design pattern support this tendency. However, in the end, the ideal usually lies somewhere in the middle. The key is to identify the areas of the contract that require flexibility and to then design them so that they allow the necessary variation.

Modularity and Reuse

Reuse of software saves time and money, and programs that are reused are often more robust and better designed all around. The individual parts that comprise Web service contracts are no exception.

As you will learn in the upcoming chapters, there is plenty of opportunity for reuse within Web service contract languages. Leveraging language features that allow for the creation of modules and provide include mechanisms can significantly streamline all parts of the service contract (and can further end up saving a lot of time that would be spent writing redundant code).

Then, of course, there is the notion of reusing parts of a contract across different contracts. This is where standardization-related patterns, such as Canonical Schema, Schema Centralization, and Policy Centralization, prove extremely valuable in not only getting more ROI out of the service contract code, but—more importantly—ensuring that these key parts are consistent across all the service contracts that establish a federated endpoint layer within a service inventory.

As with anything, though, overuse of something can lead to the opposite of the intended effect. XML language modules and centralization techniques need to be carefully chosen. Over-modularizing code within or across service contracts will inevitably lead to an inhibitive and complex architecture.

NOTE

Ultimately, one of the biggest challenges to achieving reuse is not related to technology, but to organizational governance. This is an on-going area that is addressed by all levels of service-orientation and SOA and will be covered in the upcoming book *SOA Governance*.

Clarity and Ease of Processing

With their hierarchical structure, larger-sized Web service contracts can quickly become unwieldy. Although they are intended to be read by machines, it does not mean that it doesn't matter if they are difficult to interpret by humans (especially considering that you may not want to rely on auto-generation tools to create and maintain them).

People from both the service provider and consumer sides will need to discover, understand, and work with these contracts, and any misunderstandings will introduce risk. Furthermore, overly complex contracts lead to increased governance effort and a higher cost of ownership.

Of course contracts have a job to do and sometimes this job requires that they be designed to a degree of complexity and sophistication. However, never lose sight of the fact that others will end up having to use (and perhaps evolve) what you are building today. Good contract design requires that you strive for clarity and avoid complexity or ambiguity wherever possible.

Here are some guidelines:

- *Consistent Design* – Use the same name for parts of the contract that mean the same thing, and different names for parts that are conceptually different. Developing a naming standard (as per the Canonical Expression pattern) may be the only way to guarantee that contracts are consistent in how they express themselves. This includes choosing names that are human-readable and, ideally, provide a business context. Also, when possible, use a consistent order and structure for elements that tend to be reused.
- *Limited Depth* – Some structural elements can be very useful, but you should not introduce unnecessary constraints or nested levels into the data structures that underlie your contract types. As per the Service Abstraction principle, try to define a balanced level of constraint granularity.

- *Good Documentation* – Simply providing a name and a structure for elements is not enough for the contract to be communicative to a wide range of project team members. Use annotations to further document elements and types, as per the Service Discoverability principle. (There are tools that will generate human-readable documentation from schema annotations.)

SUMMARY OF KEY POINTS

- The use of auto-generation tools can make it challenging to author standardized Web service contracts.
- Granularity, reusability, and clarity are among the common Web service contract design considerations.

Index

B

D

F

G

H

I—J

K—L

M

N

O

P

Q—R

S

T

U

V

W

X–Z