Chapter 20

Versioning Fundamentals

20.1 Basic Concepts and Terminology
20.2 Versioning and Compatibility
20.3 Version Identifiers
20.4 Versioning Strategies
After a Web service contract is deployed, consumer programs will naturally begin forming dependencies on it. When we are subsequently forced to make changes to the contract, we need to figure out:

• whether the changes will negatively impact existing (and potentially future) service consumers

• how changes that will and will not impact consumers should be implemented and communicated

These issues result in the need for versioning. Anytime you introduce the concept of versioning into an SOA project, a number of questions will likely be raised, for example:

• What exactly constitutes a new version of a service contract? What’s the difference between a major and minor version?

• What do the parts of a version number indicate?

• Will the new version of the contract still work with existing consumers that were designed for the old contract version?

• Will the current version of the contract work with new consumers that may have different data exchange requirements?

• What is the best way to add changes to existing contracts while minimizing the impact on consumers?

• Will we need to host old and new contracts at the same time? If yes, for how long?

The upcoming chapters address these questions and provide a set of options for solving common versioning problems. This chapter begins by covering some basic concepts, terminology, and strategies specific to service contract versioning in preparation for what’s ahead.
20.1 Basic Concepts and Terminology

The Scope of a Version

As we’ve established many times over in this book, a Web service contract can be comprised of several individual documents and definitions that are linked and assembled together to form a complete technical interface.

For example, a given Web service contract can consist of:

- one (sometimes more) WSDL definitions
- one (usually more) XML Schema definitions
- some (sometimes no) WS-Policy definitions

Furthermore, each of these definition documents can be shared by other Web service contracts.

For example:

- a centralized XML Schema definition will commonly be used by multiple WSDL definitions
- a centralized WS-Policy definition will commonly be applied to multiple WSDL definitions
- an abstract WSDL description can be imported by multiple concrete WSDL descriptions or vice versa

So when we say that we’re creating a new version of a contract, what exactly are we referring to?
Of all the different parts of a Web service contract, the part that establishes the fundamental technical interface is the abstract description of the WSDL definition. This represents the core of a Web service contract and is then further extended and detailed through schema definitions, policy definitions, and one or more concrete WSDL descriptions.

When we need to create a new version of a Web service contract, we can therefore assume that there has been a change in the abstract WSDL description or one of the contract documents that relates to the abstract WSDL description. How the different constructs of a WSDL can be versioned is covered in Chapter 21.

The Web service contract content commonly subject to change is the XML schema content that provides the types for the abstract description’s message definitions. Chapter 22 explores the manner in which the underlying schema definitions for messages can be changed and evolved.

Finally, the one other contract-related technology that can still impose versioning requirements but is less likely to do so simply because it is a less common part of Web service contracts is WS-Policy. How policies in general relate to contract versioning is explained as part of the advanced topics in Chapter 23.

Fine and Coarse-Grained Constraints

Versioning changes are generally related to the increase or reduction of the quantity or granularity of constraints. Therefore, let’s briefly recap the meaning of the term constraint granularity in relation to a type definition.

Note the highlighted parts of the following example:

```xml
<xsd:element name="LineItem" type="LineItemType"/>
<xsd:complexType name="LineItemType">
  <xsd:sequence>
    <xsd:element name="productID" type="xsd:string"/>
    <xsd:element name="productName" type="xsd:string"/>
    <xsd:any minOccurs="0" maxOccurs="unbounded" namespace="##any" processContents="lax"/>
  </xsd:sequence>
  <xsd:anyAttribute namespace="##any"/>
</xsd:complexType>

Example 20.1
A complexType construct containing fine and coarse-grained constraints.
```
20.2 Versioning and Compatibility

As indicated by the bolded text, there are elements with specific names and data types that represent parts of the message definition with a fine level of constraint granularity. All of the message instances (the actual XML documents that will be created based on this structure) must conform to these constraints in order to be considered valid (which is why these are considered the absolute "minimum" constraints).

The red text shows the element and attribute wildcards also contained by this complex type. These represent parts of the message definition with an extremely coarse level of constraint granularity in that messages do not need to comply to these parts of the message definition at all.

The use of the terms “fine-grained” and “coarse-grained” is highly subjective. What may be a fine-grained constraint in one contract may not be in another. The point is to understand how these terms can be applied when comparing parts of a message definition or when comparing different message definitions with each other.

20.2 Versioning and Compatibility

The number one concern when developing and deploying a new version of a service contract is the impact it will have on other parts of the enterprise that have formed or will form dependencies on it. This measure of impact is directly related to how compatible the new contract version is with the old version and its surroundings in general.

This section establishes the fundamental types of compatibility that relate to the content and design of new contract versions and also tie into the goals and limitations of different versioning strategies that we introduce at the end of this chapter.

Backwards Compatibility

A new version of a Web service contract that continues to support consumer programs designed to work with the old version is considered backwards-compatible. From a design perspective, this means that the new contract has not changed in such a way that it can impact existing consumer programs that are already using the contract.

A simple example of a backwards-compatible change is the addition of a new operation to an existing WSDL definition:
Example 20.2
The addition of a new operation represents a common backwards-compatible change.

In this example we’re borrowing the abstract description of the Purchase Order service that was initially built at the end of Chapter 7. By adding a brand new operation, we are creating a new version of the contract, but this change is backwards-compatible and will not impact any existing consumers.

An example of a change made to a schema for a message definition that is backwards compatible is the addition of an optional element:
20.2 Versioning and Compatibility

NOTE
Both the Flexible and Loose versioning strategies explained at the end of this chapter support backwards compatibility.

Forwards Compatibility

When a Web service contract is designed in such a manner so that it can support a range of future consumer programs, it is considered to have an extent of *forwards compatibility*. This means that the contract can essentially accommodate how consumer programs will evolve over time.
The most common means by which forwards compatibility is attempted in message definitions is through the use of wildcards:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po"
    xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:any namespace="#any" processContents="lax"
          minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:anyAttribute namespace="#any"/>
  </xsd:complexType>
</xsd:schema>
```

Example 20.4
To support forwards compatibility within a message definition generally requires the use of XML Schema wildcards.

In this example, the `xsd:any` and `xsd:anyAttribute` elements are added to allow for a range of unknown elements and data to be accepted by the Web service contract. In other words, the schema is being designed in advance to accommodate unforeseen changes in the future. Chapter 22 explains in detail how wildcards can be used in support of forwards compatibility.

There are limited options in support of forwards compatibility when it comes to WSDL definitions. These are discussed at the end of Chapter 21.

It is important to understand that forwards compatibility is by no means an exact science. A service with a forwards-compatible contract will often not be able to process all message content. It’s contract is simply designed to accept a broader range of data unknown at the time of its design.

NOTE
Forwards compatibility forms the basis of the Loose versioning strategy that is explained at the end of this chapter.
20.2 Versioning and Compatibility

Compatible Changes

When we make a change to a Web service contract that does not negatively affect its existing consumers, then the change itself is considered a compatible change.

NOTE

In this book, the term "compatible change" refers to backwards compatibility by default. When used in reference to forwards compatibility it is further qualified as a forwards-compatible change.

A simple example of a compatible change is when we set the minOccurs attribute of an element from “1” to “0”, effectively turning a required element into an optional one, as shown here:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po"
    xmlns="http://actioncon.com/schema/po">
    <xsd:element name="LineItem" type="LineItemType"/>
    <xsd:complexType name="LineItemType">
        <xsd:sequence>
            <xsd:element name="productID" type="xsd:string"/>
            <xsd:element name="productName" type="xsd:string"
                minOccurs="0"/>
            <xsd:element name="available" type="xsd:boolean"
                minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

The default value of the minOccurs attribute is "1". Therefore because this attribute was previously absent from the productName element declaration, it was considered a required element. Adding the minOccurs="0" setting turns it into an optional element, resulting in a compatible change.

This type of change will not impact existing consumer programs that are used to sending the element value to the Web service, nor will it affect future consumers that can be designed to optionally send that element.

Another example of a compatible change was provided earlier in Example 20.3, when we first added the optional available element declaration. Even though we extended the type with a whole new element, because it is optional it is considered a compatible change.
Here is a list of common compatible changes that we will be discussing in the upcoming chapters:

- adding a new WSDL operation definition and associated message definitions (Chapter 21)
- adding a new WSDL port type definition and associated operation definitions (Chapter 21)
- adding new WSDL binding and service definitions (Chapter 21)
- adding a new optional XML Schema element or attribute declaration to a message definition (Chapter 22)
- reducing the constraint granularity of an XML Schema element or attribute of a message definition type (Chapter 22)
- adding a new XML Schema wildcard to a message definition type (Chapter 22)
- adding a new optional WS-Policy assertion (Chapter 23)
- adding a new WS-Policy alternative (Chapter 23)

We will also be exploring techniques whereby changes that are not normally compatible can still be implemented as compatible changes.

**NOTE**

Compatible Change is also the name of a versioning design pattern that is based on the techniques described in this book for preserving backwards compatibility when modifying a service contract. See Appendix E for a description of this pattern.

**Incompatible Changes**

If after a change a contract is no longer compatible with consumers, then it is considered to have received an *incompatible change*. These are the types of changes that can break an existing contract and therefore impose the most challenges when it comes to versioning.

**NOTE**

The term "incompatible change" indicates an absence of backwards compatibility. When referring to incompatible changes that affect forwards compatibility, this term is qualified as *forwards incompatible change*.
20.2 Versioning and Compatibility

Going back to our example, if we set an element’s minOccurs attribute from “0” to any number above zero, then we are introducing an incompatible change:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"
        minOccurs="3"/>
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="3"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 20.6
Incrementing the minOccurs attribute value of any established element declaration is automatically an incompatible change.

What was formerly an optional element is now required. This will certainly affect existing consumers that are not designed to comply with this new constraint, because adding a new required element introduces a mandatory constraint upon the contract.

Common incompatible changes that are explained in the next set of chapters include:

- renaming an existing WSDL operation definition
- removing an existing WSDL operation definition
- changing the MEP of an existing WSDL operation definition
- adding a fault message to an existing WSDL operation definition
- adding a new required XML Schema element or attribute declaration to a message definition
- increasing the constraint granularity of an XML Schema element or attribute declaration of a message definition
- renaming an optional or required XML Schema element or attribute in a message definition
610  Chapter 20: Versioning Fundamentals

- removing an optional or required XML Schema element or attribute or wildcard from a message definition
- adding a new required WS-Policy assertion or expression
- adding a new ignorable WS-Policy expression (most of the time)

Incompatible changes cause most of the challenges with Web service contract versioning.

20.3 Version Identifiers

One of the most fundamental design patterns related to Web service contract design is the Version Identification pattern. It essentially advocates that version numbers should be clearly expressed, not just at the contract level, but right down to the versions of the schemas that underlie the message definitions.

The first step to establishing an effective versioning strategy is to decide on a common means by which versions themselves are identified and represented within Web service contracts.

Versions are almost always communicated with version numbers. The most common format is a decimal, followed by a period and then another decimal, as shown here:

```xml
version="2.0"
```

Sometimes, you will see additional period + decimal pairs that lead to more detailed version numbers like this:

```xml
version="2.0.1.1"
```

The typical meaning associated with these numbers is the measure or significance of the change. Incrementing the first decimal generally indicates a major version change (or upgrade) in the software, whereas decimals after the first period usually represent various levels of minor version changes.

From a compatibility perspective, we can associate additional meaning to these numbers. Specifically, the following convention has emerged in the industry:

- A minor version is expected to be backwards compatible with other minor versions associated with a major version. For example, version 5.2 of a program should be fully backwards compatible with versions 5.0 and 5.1.
20.3 Version Identifiers

- A major version is generally expected to break backwards compatibility with programs that belong to other major versions. This means that program version 5.0 is not expected to be backwards compatible with version 4.0.

This convention of indicating compatibility through major and minor version numbers is referred to as the compatibility guarantee. Another approach, known as “amount of work,” uses version numbers to communicate the effort that has gone into the change. A minor version increase indicates a modest effort, and a major version increase predictably represents a lot of work.

These two conventions can be combined and often are. The result is often that version numbers continue to communicate compatibility as explained earlier, but they sometimes increment by several digits, depending on the amount of effort that went into each version.

There are various syntax options available to express version numbers. For example, you may have noticed that the declaration statement that begins an XML document can contain a number that expresses the version of the XML specification being used:

```xml
<?xml version="1.0"?>
```

That same version attribute can be used with the root xsd:schema element, as follows:

```xml
<xsd:schema version="2.0" ...>
```

You can further create a custom variation of this attribute by assigning it to any element you define (in which case you are not required to name the attribute “version”).

```xml
<LineItem version="2.0">
```

An alternative custom approach is to embed the version number into a namespace, as shown here:

```xml
<LineItem xmlns="http://actioncon.com/schema/po/v2">
```

Note that it has become a common convention to use date values in namespaces when versioning XML schemas, as follows:

```xml
```

In this case, it is the date of the change that acts as the version identifier. In order to keep the expression of XML Schema definition versions in alignment with WSDL
definition versions, we use version numbers instead of date values in the examples throughout the upcoming chapters. However, when working in an environment where XML Schema definitions are separately owned as part of an independent data architecture, it is not uncommon for schema versioning identifiers to be different from those used by WSDL definitions.

Regardless of which option you choose, it is important to consider the Canonical Versioning pattern that dictates that the expression of version information must be standardized across all service contracts within the boundary of a service inventory. In larger environments, this will often require a central authority that can guarantee the linearity, consistency, and description quality of version information. These types of conventions carry over into how service termination information is expressed (as further explored in Chapter 23).

<table>
<thead>
<tr>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Of course you may also be required to work with third-party schemas and WSDL definitions that may already have implemented their own versioning conventions. In this case, the extent to which the Canonical Versioning pattern can be applied will be limited.</td>
</tr>
</tbody>
</table>

## 20.4 Versioning Strategies

There is no one versioning approach that is right for everyone. Because versioning represents a governance-related phase in the overall lifecycle of a service, it is a practice that is subject to the conventions, preferences, and requirements that are distinct to any enterprise.

Even though there is no de facto versioning technique for the WSDL, XML Schema, and WS-Policy content that comprises Web service contracts, a number of common and advocated versioning approaches have emerged, each with its own benefits and tradeoffs.

In this chapter we’re going to single out the following three known strategies:

- **Strict** – Any compatible or incompatible changes result in a new version of the service contract. This approach does not support backwards or forwards compatibility.

- **Flexible** – Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility but not forwards compatibility.
20.4 Versioning Strategies

• *Loose* – Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility and forwards compatibility.

These strategies are explained individually in the upcoming sections and referenced throughout the remaining chapters.

**The Strict Strategy (New Change, New Contract)**

The simplest approach to Web service contract versioning is to require that a new version of a contract be issued whenever any kind of change is made to any part of the contract.

This is commonly implemented by changing the target namespace value of a WSDL definition (and possibly the XML Schema definition) every time a compatible or incompatible change is made to the WSDL, XML Schema, or WS-Policy content related to the contract. Namespaces are used for version identification instead of a version attribute because changing the namespace value automatically forces a change in all consumer programs that need to access the new version of the schema that defines the message types.

This “super-strict” approach is not really that practical, but it is the safest and sometimes warranted when there are legal implications to Web service contract modifications, such as when contracts are published for certain inter-organization data exchanges. Because both compatible and incompatible changes will result in a new contract version, this approach supports neither backwards nor forwards compatibility.

*Pros and Cons*

The benefit of this strategy is that you have full control over the evolution of the service contract, and because backwards and forwards compatibility are intentionally disregarded, you do not need to concern yourself with the impact of any change in particular (because all changes effectively break the contract).

On the downside, by forcing a new namespace upon the contract with each change, you are guaranteeing that all existing service consumers will no longer be compatible with any new version of the contract. Consumers will only be able to continue communicating with the Web service while the old contract remains available alongside the new version or until the consumers themselves are updated to conform to the new contract.
Therefore, this approach will increase the governance burden of individual services and will require careful transitioning strategies. Having two or more versions of the same service co-exist at the same time can become a common requirement for which the supporting service inventory infrastructure needs to be prepared.

**The Flexible Strategy (Backwards Compatibility)**

A common approach used to balance practical considerations with an attempt at minimizing the impact of changes to Web service contracts is to allow compatible changes to occur without forcing a new contract version, while not attempting to support forwards compatibility at all.

This means that any backwards-compatible change is considered safe in that it ends up extending or augmenting an established contract without affecting any of the service’s existing consumers. A common example of this is adding a new operation to a WSDL definition or adding an optional element declaration to a message’s schema definition.

As with the Strict strategy, any change that breaks the existing contract does result in a new contract version, usually implemented by changing the target namespace value of the WSDL definition and potentially also the XML Schema definition.

**Pros and Cons**

The primary advantage to this approach is that it can be used to accommodate a variety of changes while consistently retaining the contract’s backwards compatibility. However, when compatible changes are made, these changes become permanent and cannot be reversed without introducing an incompatible change. Therefore, a governance process is required during which each proposed change is evaluated so that contracts do not become overly bloated or convoluted. This is an especially important consideration for agnostic services that are heavily reused.

**The Loose Strategy (Backwards and Forwards Compatibility)**

As with the previous two approaches, this strategy requires that incompatible changes result in a new service contract version. The difference here is in how service contracts are initially designed.

Instead of accommodating known data exchange requirements, special features from the WSDL, XML Schema, and WS-Policy languages are used to make parts of the
20.4 Versioning Strategies

contract intrinsically extensible so that they remain able to support a broad range of future, unknown data exchange requirements.

For example:

• The anyType attribute value provided by the WSDL 2.0 language allows a message to consist of any valid XML document.
• XML Schema wildcards can be used to allow a range of unknown data to be passed in message definitions.
• Ignorable policy assertions can be defined to communicate service characteristics that can optionally be acknowledged by future consumers.

These and other features related to forwards compatibility are discussed in upcoming chapters.

Pros and Cons

The fact that wildcards allow undefined content to be passed through Web service contracts provides a constant opportunity to further expand the range of acceptable message element and data content. On the other hand, the use of wildcards will naturally result in vague and overly coarse service contracts that place the burden of validation on the underlying service logic.

NOTE

All three strategies will be referenced in upcoming chapters as we explore how versioning can be accomplished with the WSDL, XML Schema, and WS-Policy languages.

Summary Table

Provided here is a table that broadly summarizes how the three strategies compare based on three fundamental characteristics.

The three characteristics used in this table to form the basis of this comparison are as follows:

• Strictness – The rigidity of the contract versioning options. The Strict approach clearly is the most rigid in its versioning rules, while the Loose strategy provides the broadest range of versioning options due to its reliance on wildcards.
• **Governance Impact** – The amount of governance burden imposed by a strategy. Both Strict and Loose approaches increase governance impact but for different reasons. The Strict strategy requires the issuance of more new contract versions, which impacts surrounding consumers and infrastructure, while the Loose approach introduces the concept of unknown message sets that need to be separately accommodated through custom programming.

• **Complexity** – The overall complexity of the versioning process. Due to the use of wildcards and unknown message data, the Loose strategy has the highest complexity potential, while the straightforward rules that form the basis of the Strict approach make it the simplest option.

Throughout this comparison, the Flexible strategy provides an approach that represents a consistently average level of strictness, governance effort, and overall complexity.

<table>
<thead>
<tr>
<th></th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Strict</td>
</tr>
<tr>
<td>Strictness</td>
<td>high</td>
</tr>
<tr>
<td>Governance Impact</td>
<td>high</td>
</tr>
<tr>
<td>Complexity</td>
<td>low</td>
</tr>
</tbody>
</table>

Table 20.1
A general comparison of the three versioning strategies.
Chapter 21

Versioning WSDL Definitions

21.1 Version Identifiers and WSDL Definitions
21.2 Versioning Operation Definitions
21.3 Versioning Port Type Definitions
21.4 Versioning Concrete Descriptions
21.5 WSDL Definitions and Forwards Compatibility
Whereas changes to message definition types and policies are often supplementary and less evident, alterations made to a WSDL document will have the most visible impact.

The following sections document a series of common change types for definitions within both abstract and concrete descriptions with an emphasis on the Strict and Flexible versioning approaches. Compared to the schema versioning scenarios covered in Chapter 22, there is not as much opportunity to support forwards compatibility. Hence, techniques related to the Loose approach are summarized at the end of this chapter in the WSDL Definitions and Forwards Compatibility section.

21.1 Version Identifiers and WSDL Definitions

As with any contract-related document, a WSDL definition can be subject to compatible and incompatible changes. Based on your versioning strategy and conventions, you can decide how to best apply the identifiers explained in Chapter 20 to express new WSDL definition versions. In the upcoming sections (and throughout the remaining versioning chapters), we will follow the "significance of change" approach to version numbering, plus we will incorporate the use of namespaces.

Unlike the XML Schema xsd:schema element, the WSDL definitions element does not provide a built-in version attribute for us to use. We'll therefore use the WSDL documentation element to store the version number instead.

Here are the conventions for our versioning scheme:

- Minor and major contract version numbers will be expressed using a documentation element that follows the opening definitions element. The version numbers will be displayed after the word “Version,” as follows:

  `<documentation>Version 1.0</documentation>`

- Major version numbers will be appended to the WSDL definition’s target namespace and prefixed with a “v” as shown here: http://actioncon.com/contract/po/v2
21.1 Version Identifiers and WSDL Definitions

- The exception to the preceding rule is when the first version of a WSDL definition is released. In this case, no version number is added to the namespace.

- A compatible change increments the minor version number and does not change the WSDL definition’s target namespace.

- An incompatible change increments the major version number and results in a new target namespace for the WSDL definition.

The following example shows how version 2.1 of a WSDL definition would be expressed, based on our version identification strategy:

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v2"
xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:tns="http://actioncon.com/contract/po/v2"
xmlns:po="http://actioncon.com/schema/po"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<documentation>Version 2.1</documentation>
...
</definitions>
```

Example 21.1
The `definitions` element with a target namespace that indicates that this is the second version of the Web service contract.

Note that this approach to version identification is used regardless of where compatible or incompatible changes originated. We are not limiting ourselves to only expressing version numbers in the WSDL header area. In fact, we are usually required to add further version identification to the part of the Web service contract that triggered the change.

For example, an incompatible change to the XML Schema type associated with a message definition will require us to increment version numbers within the schema in addition to the major version number in the WSDL definition’s target namespace and documentation element. Also as we will shortly explore, it sometimes makes sense to further embed version numbers within the names of specific WSDL elements.
Base Example

The upcoming sections add a variety of changes to the existing Web service contract for the Purchase Order service. The majority of these changes impact the abstract description.

Provided here is a base example of the Purchase Order WSDL definition containing the original abstract description that we designed in prior chapters plus the new versioning identifiers explained in the previous section. Our base example therefore represents version 2.1 of the Purchase Order service contract. Upcoming changes to this example will be highlighted using red or bolded text.

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v2"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:tns=" http://actioncon.com/contract/po/v2"
    xmlns:po="http://actioncon.com/schema/po"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <documentation>Version 2.1</documentation>
    <types>
        <xsd:schema>
            <xsd:import namespace="http://actioncon.com/schema/po"
                schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
        </xsd:schema>
    </types>
    <message name="msgSubmitOrderRequest">
        <part name="PurchaseOrder" element="po:purchaseOrder"/>
    </message>
    <message name="msgSubmitOrderResponse">
        <part name="Acknowledgement" element="po:acknowledgement"/>
    </message>
    <message name="msgCheckOrderRequest">
        <part name="PONumber" element="po:poNumber"/>
    </message>
    <message name="msgCheckOrderResponse">
        <part name="Status" element="po:status"/>
    </message>
    <message name="msgChangeOrderRequest">
        <part name="PurchaseOrder" element="po:purchaseOrder"/>
    </message>
    <message name="msgChangeOrderResponse">
        <part name="Acknowledgement" element="po:acknowledgement"/>
    </message>
</definitions>
```
21.1 Version Identifiers and WSDL Definitions

SUMMARY OF KEY POINTS

- The version identification approach used in the upcoming examples requires that major and minor numbers are displayed in the documentation element and that major numbers are incorporated in the WSDL target namespace.

- The following examples will demonstrate versioning scenarios that modify a common base example.

```
<message name="msgCancelOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgCancelOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>

<portType name="ptPurchaseOrder">
  <operation name="opSubmitOrder">
    <input message="tns:msgSubmitOrderRequest"/>
    <output message="tns:msgSubmitOrderResponse"/>
  </operation>
  <operation name="opCheckOrderStatus">
    <input message="tns:msgCheckOrderRequest"/>
    <output message="tns:msgCheckOrderResponse"/>
  </operation>
  <operation name="opChangeOrder">
    <input message="tns:msgChangeOrderRequest"/>
    <output message="tns:msgChangeOrderResponse"/>
  </operation>
  <operation name="opCancelOrder">
    <input message="tns:msgCancelOrderRequest"/>
    <output message="tns:msgCancelOrderResponse"/>
  </operation>
</portType>
```

Example 21.2
The abstract description of version 2.1 of the Purchase Order WSDL definition prior to being impacted by further changes that impose additional versioning requirements.
21.2 Versioning Operation Definitions

The majority of changes on the WSDL definition level are generally centered on operations. This, the largest section in this chapter, covers the following range of common operation-related changes:

- adding a new operation
- renaming an existing operation
- removing an existing operation
- changing the MEP of an existing operation
- adding a fault message to an existing operation

As mentioned at the beginning of this chapter, these changes are discussed for both the Strict and Flexible versioning approaches. Additional methods in support of the Loose versioning strategy are discussed at the end of this chapter.

Adding a New Operation

Appending a WSDL Definition with an Operation as a Compatible Change (Flexible)

When a WSDL definition is already implemented and in use, consumer programs will have likely formed dependencies on existing operation definitions. Extending the contract by adding a new operation definition will not impact these dependencies and is therefore considered a backwards compatible change by default.

As shown by the highlighted text in the following example, the newly added operation construct and its related message constructs do not affect any other part of the WSDL definition, except for the version number:

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v2" xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://actioncon.com/contract/po/v2"
xmlns:po="http://actioncon.com/schema/po"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<documentation>Version 2.2</documentation>
<types>
<xsd:schema>
  <xsd:import namespace="http://actioncon.com/schema/po"
    schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
</xsd:schema>
</types>
</definitions>
```
21.2 Versioning Operation Definitions

```xml
<message name="msgSubmitOrderRequest">
    <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>

<message name="msgSubmitOrderResponse">
    <part name="Acknowledgement" element="po:acknowledgement"/>
</message>

<message name="msgCheckOrderRequest">
    <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgCheckOrderResponse">
    <part name="Status" element="po:status"/>
</message>

<message name="msgChangeOrderRequest">
    <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>

<message name="msgChangeOrderResponse">
    <part name="Acknowledgement" element="po:acknowledgement"/>
</message>

<message name="msgCancelOrderRequest">
    <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgCancelOrderResponse">
    <part name="Acknowledgement" element="po:acknowledgement"/>
</message>

<message name="msgGetOrderRequest">
    <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgGetOrderResponse">
    <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>

<portType name="ptPurchaseOrder">
    <operation name="opSubmitOrder">
        <input message="tns:msgSubmitOrderRequest"/>
        <output message="tns:msgSubmitOrderResponse"/>
    </operation>
    <operation name="opCheckOrderStatus">
        <input message="tns:msgCheckOrderRequest"/>
        <output message="tns:msgCheckOrderResponse"/>
    </operation>
    <operation name="opChangeOrder">
        <input message="tns:msgChangeOrderRequest"/>
        <output message="tns:msgChangeOrderResponse"/>
    </operation>
    <operation name="opCancelOrder">
        <input message="tns:msgCancelOrderRequest"/>
    </operation>
</portType>
```
The manner in which the new operation was added in the preceding example supports the Flexible versioning approach in that it allows a compatible change to be made without forcing a new major contract version.

**Forcing a New Contract Version when Adding an Operation (Strict)**

When following the Strict approach, the addition of the new operation and message definitions will increment the contract version number to “3.0” and also change the WSDL target namespace value to reflect the new version, as follows:

```xml
<definitions name="PurchaseOrder"
targetNamespace= "http://actioncon.com/contract/po/v3"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmns:tns="http://actioncon.com/contract/po/v3"
xmns:po="http://actioncon.com/schema/po"
xmns:xsd="http://www.w3.org/2001/XMLSchema">
  <documentation>Version 3.0</documentation>
  ...
</definitions>
```

**Example 21.4**
The same changes made to the WSDL definition in Example 21.3 force a new contract version when carrying out the Strict approach.

**Renaming an Existing Operation**

If the value of the name attribute of an existing operation element needs to be changed after the WSDL document has been deployed, then this will clearly impact any consumers that have already been designed to use this operation.
21.2 Versioning Operation Definitions

There are two common ways of handling this type of change:

1. Force a new major version of the contract.
2. Add the renamed operation to the existing contract.

Because renaming an operation represents an incompatible change, the first method is compliant with both Strict and Flexible versioning strategies. The second technique proposes a way of renaming an operation as a compatible change and is therefore intended for the Flexible approach (although it can also technically be used as part of a Strict approach).

Forcing a New Major Contract Version (Strict, Flexible)

If the existing operation name must be modified, then the contract is subjected to an incompatible change that will require that a new major version of the contract be created, as follows:

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v3"
xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:tns="http://actioncon.com/contract/po/v3"
xmlns:po="http://actioncon.com/schema/po"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<documentation>Version 3.0</documentation>
<types>
<xsd:schema>
<xsd:import namespace="http://actioncon.com/schema/po"
schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
</xsd:schema>
</types>
<message name="msgSubmitOrdersRequest">
<part name="PurchaseOrder" element="po:purchaseOrders"/>
</message>
<message name="msgSubmitOrdersResponse">
<part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgCheckOrderRequest">
<part name="PONumber" element="po:poNumber"/>
</message>
<message name="msgCheckOrderResponse">
<part name="Status" element="po:status"/>
```
</message>
<message name="msgChangeOrderRequest">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
<message name="msgChangeOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgCancelOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>
<message name="msgCancelOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgGetOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>
<message name="msgGetOrderResponse">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
<portType name="ptPurchaseOrder">
  <operation name="opSubmitOrders">
    <input message="tns:msgSubmitOrdersRequest"/>
    <output message="tns:msgSubmitOrdersResponse"/>
  </operation>
  <operation name="opCheckOrderStatus">
    <input message="tns:msgCheckOrderRequest"/>
    <output message="tns:msgCheckOrderResponse"/>
  </operation>
  <operation name="opChangeOrder">
    <input message="tns:msgChangeOrderRequest"/>
    <output message="tns:msgChangeOrderResponse"/>
  </operation>
  <operation name="opCancelOrder">
    <input message="tns:msgCancelOrderRequest"/>
    <output message="tns:msgCancelOrderResponse"/>
  </operation>
  <operation name="opGetOrder">
    <input message="tns:msgGetOrderRequest"/>
    <output message="tns:msgGetOrderResponse"/>
  </operation>
</portType>
</definitions>

Example 21.5
The opSubmitOrder operation name is changed.
In this example, the \texttt{opSubmitOrder} operation is renamed to \texttt{"opSubmitOrders"} to indicate that it has been redeveloped to support the simultaneous submissions of multiple purchase orders at the same time.

This could easily impact the message definitions and their underlying schema types that now must support messages that can contain one or more entire purchase order documents. In this case, the imported schema itself may also need to undergo a versioning change reflected in the new XML Schema target namespace shown here:

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v3"
xmlns="http://schemas.xmlsoap.org/wsd1/"
xmlns:tns="http://actioncon.com/contract/po/v3"
xmlns:po="http://actioncon.com/schema/po/v3"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <documentation>Version 3.0</documentation>
  <types>
    <xsd:schema>
      <xsd:import namespace="http://actioncon.com/schema/po/v3"
                    schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
    </xsd:schema>
  </types>
</definitions>
```

Example 21.6
The target namespace of the imported XML Schema definition is changed in response to incompatible changes made to the underlying schema content.

\textit{Extending the Contract with the Renamed Operation (Flexible)}

Depending on the reasons behind the name change, it might make sense to simply preserve the existing contract and add a new operation definition with the new name.

In the following variation of the previous example, the \texttt{opSubmitOrders} operation is added alongside the original \texttt{opSubmitOrder} operation:

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v2"
xmlns="http://schemas.xmlsoap.org/wsd1/"
```
<documentation>Version 2.2</documentation>
<types>
  <xsd:schema>
    <xsd:import namespace="http://actioncon.com/schema/po"
      schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
  </xsd:schema>
</types>
<message name="msgSubmitOrderRequest">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
<message name="msgSubmitOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgSubmitOrdersRequest">
  <part name="PurchaseOrder" element="po:purchaseOrders"/>
</message>
<message name="msgSubmitOrdersResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgCheckOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>
<message name="msgCheckOrderResponse">
  <part name="Status" element="po:status"/>
</message>
<message name="msgChangeOrderRequest">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
<message name="msgChangeOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgCancelOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>
<message name="msgCancelOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgGetOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>
<message name="msgGetOrderResponse">
In this scenario, the addition of the new operation actually demonstrates the application of the Contract Denormalization design pattern that allows operations with overlapping functionality to exist in the same service contract.

**Example 21.7**

A new `opSubmitOrders` operation is added as a compatible change.

NOTE

The aforementioned technique can be applied when following the Strict approach as long as it forces a new contract version. In this case, it is equivalent to adding a new operation as explained previously in the *Forcing a New Contract Version when Adding an Operation (Strict)* section.
Operation Termination

In the case that the renamed operation represents exactly the same underlying functionality as the original operation, this approach can be considered “transitionary.” It is similar to forcing a new version of a contract and then hosting both old and new contracts simultaneously, in that the port type will host old and new versions of the same operation.

As when having separate contracts, a retirement strategy will likely be required for the original operation. In the following example, an annotation is added to communicate the target termination date:

```xml
<portType name="ptPurchaseOrder">
  <documentation>
    opSubmitOrder is Scheduled for Termination on 01/12
  </documentation>
  <operation name="opSubmitOrder">
    <input message="tns:msgSubmitOrderRequest"/>
    <output message="tns:msgSubmitOrderResponse"/>
  </operation>
  ...
</portType>
```

Example 21.8

The opSubmitOrder operation construct is annotated to indicate its termination date.

NOTE

Techniques for expressing various types of termination information are provided in Chapter 23.

Removing an Existing Operation

The actual removal of an operation definition from a contract will predictably impact consumers that have formed dependencies on it because after the operation has been removed, calls to that Web service operation will predictably fail.

As with renaming an operation, there are two common approaches to accommodate this change:

1. The operation definition is deleted, forcing a new major version of the contract.
2. The operation definition is preserved as a functional stub.
21.2 Versioning Operation Definitions

Unlike renaming an operation, we do not have an option whereby the operation can be removed from the contract as a compatible change. Therefore, both techniques impose an incompatible change, regardless of whether we are following a Strict or Flexible approach.

Removing the Operation and Forcing a New Major Contract Version (Strict, Flexible)

The operation construct and its associated message constructs are removed, and a whole new WSDL definition is released, as indicated by the incremented major version numbers:

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v3"
xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:tns="http://actioncon.com/contract/po/v3"
xmlns:po="http://actioncon.com/schema/po"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<documentation>Version 3.0</documentation>
<types>
  <xsd:schema>
    <xsd:import namespace="http://actioncon.com/schema/po"
       schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
  </xsd:schema>
</types>
<message name="msgSubmitOrderRequest">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
<message name="msgSubmitOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>

<!-- msgCheckOrderRequest Removed 01/12 -->

<!-- msgCheckOrderResponse Removed 01/12 -->

<message name="msgChangeOrderRequest">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
<message name="msgChangeOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgCancelOrderRequest">
```
The highlighted comments indicate when the operation-related content was removed from the contract. This may be helpful as a historical reference for consumer designers.

Turning the Operation into a Functional Stub and Forcing a New Major Contract Version (Strict, Flexible)

One way to delete the functionality of the operation while reducing (but not eliminating) the impact upon consumers is to turn the original operation definition into a stub:
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v3"
xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:tns="http://actioncon.com/contract/po/v3"
xmlns:po="http://actioncon.com/schema/po"
xmlns:er="http://actioncon.com/schema/po/errors"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<documentation>Version 3.0</documentation>
<types>
<xsd:schema>
<xsd:import namespace="http://actioncon.com/schema/po"
schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
</xsd:schema>
<xsd:schema targetNamespace="http://actioncon.com/schema/po/errors">
<xsd:element name="statusError" type="xsd:string"/>
</xsd:schema>
</types>
<message name="msgSubmitOrderRequest">
<part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
<message name="msgSubmitOrderResponse">
<part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgCheckOrderRequest">
<part name="PONumber" element="po:poNumber"/>
</message>
<message name="msgCheckOrderResponse">
<part name="StatusError" element="er:statusError"/>
</message>
<message name="msgChangeOrderRequest">
<part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
<message name="msgChangeOrderResponse">
<part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgCancelOrderRequest">
<part name="PONumber" element="po:poNumber"/>
</message>
<message name="msgCancelOrderResponse">
<part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgGetOrderRequest">
<part name="PONumber" element="po:poNumber"/>
</message>
</definitions>
The msgCheckOrderRequest message remains unchanged, allowing the opCheckOrderStatus operation to continue accepting the same input message. But instead of returning the status value, it responds with an error message that indicates that the status value is no longer available with this operation.

The type for this message is (in this case) embedded within the types construct as a separate schema so as not to impose versioning requirements upon the purchaseOrder.xsd schema document (and also because this is considered a contract-specific message).

You can decide whether you want the error message to comply with the schema type so that it is successfully validated by the consumer, or whether it should be based on a different type in order to deliberately fail validation.
21.2 Versioning Operation Definitions

Even though this might appear to be a quasi-compatible change, the fact that the behavior and data associated with the `opCheckOrderStatus` operation have been significantly augmented will almost always make this an incompatible change, thereby requiring a new major contract version (along with a new namespace).

<table>
<thead>
<tr>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>This will also likely be a temporary measure until the operation is fully retired and then actually removed from the contract, in which case termination information can also be added.</td>
</tr>
</tbody>
</table>

Changing the MEP of an Existing Operation

Consumers form a dependency on the MEP established by the input and output message definitions associated with an operation as much as they do on the underlying types of the individual messages. Therefore, adding an input or output message to an operation or changing the order of these message definitions is considered an incompatible change that will break the contract.

**Changing the MEP and Forcing a New Major Contract Version (Strict, Flexible)**

In the following example, the `opCancelOrder` operation is changed from a request-response to a one-way MEP by removing the output message:

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v3"
xmlns="http://schemas.xmlsoap.org/wsd1/"
xmlns:tns="http://actioncon.com/contract/po/v3"
xmlns:po="http://actioncon.com/schema/po"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<documentation>Version 3.0</documentation>
<types>
  <xsd:schema>
    <xsd:import namespace="http://actioncon.com/schema/po"
      schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
  </xsd:schema>
</types>
<message name="msgSubmitOrderRequest">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
```
<message name="msgSubmitOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgCheckOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgCheckOrderResponse">
  <part name="Status" element="po:status"/>
</message>

<message name="msgChangeOrderRequest">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>

<message name="msgChangeOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>

<message name="msgCancelOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>

<!-- msgCancelOrderResponse Removed 01/12 -->

<message name="msgGetOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgGetOrderResponse">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>

<portType name="ptPurchaseOrder">
  <operation name="opSubmitOrder">
    <input message="tns:msgSubmitOrderRequest"/>
    <output message="tns:msgSubmitOrderResponse"/>
  </operation>
  <operation name="opCheckOrderStatus">
    <input message="tns:msgCheckOrderRequest"/>
    <output message="tns:msgCheckOrderResponse"/>
  </operation>
  <operation name="opChangeOrder">
    <input message="tns:msgChangeOrderRequest"/>
    <output message="tns:msgChangeOrderResponse"/>
  </operation>
  <operation name="opCancelOrder">
    <input message="tns:msgCancelOrderRequest"/>
  </operation>
  <operation name="opGetOrder">
    <input message="tns:msgGetOrderRequest"/>
    <output message="tns:msgGetOrderResponse"/>
  </operation>
</portType>
In this example, the corresponding msgCancelOrderResponse message definition is also removed, as it is no longer required. The result is an incompatible change that forces a new major contract version and a new target namespace for the WSDL definition.

**Extending the Contract with the Operation Containing the Modified MEP (Flexible)**

An alternative technique is to simply append the WSDL definition with a new operation definition that contains the new MEP. This allows the operation with the modified MEP to reside alongside the original operation, as follows:

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v2"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:tns="http://actioncon.com/contract/po/v2"
    xmlns:po="http://actioncon.com/schema/po"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <documentation>Version 2.2</documentation>
    <types>
        <xsd:schema>
            <xsd:import namespace="http://actioncon.com/schema/po"
                schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
        </xsd:schema>
    </types>
    <message name="msgSubmitOrderRequest">
        <part name="PurchaseOrder" element="po:purchaseOrder"/>
    </message>
    <message name="msgSubmitOrderResponse">
        <part name="Acknowledgement" element="po:acknowledgement"/>
    </message>
    <message name="msgCheckOrderRequest">
        <part name="PONumber" element="po:poNumber"/>
    </message>
    <message name="msgCheckOrderResponse">
        <part name="Status" element="po:status"/>
    </message>
</definitions>
```
<message name="msgChangeOrderRequest">
    <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>

<message name="msgChangeOrderResponse">
    <part name="Acknowledgement" element="po:acknowledgement"/>
</message>

<message name="msgCancelOrderRequest">
    <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgCancelOrderResponse">
    <part name="Acknowledgement" element="po:acknowledgement"/>
</message>

<message name="msgCancelOrderNotifyRequest">
    <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgGetOrderRequest">
    <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgGetOrderResponse">
    <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>

<portType name="ptPurchaseOrder">
    <operation name="opSubmitOrder">
        <input message="tns:msgSubmitOrderRequest"/>
        <output message="tns:msgSubmitOrderResponse"/>
    </operation>
    
    <operation name="opCheckOrderStatus">
        <input message="tns:msgCheckOrderRequest"/>
        <output message="tns:msgCheckOrderResponse"/>
    </operation>
    
    <operation name="opChangeOrder">
        <input message="tns:msgChangeOrderRequest"/>
        <output message="tns:msgChangeOrderResponse"/>
    </operation>
    
    <operation name="opCancelOrder">
        <input message="tns:msgCancelOrderRequest"/>
        <output message="tns:msgCancelOrderResponse"/>
    </operation>
    
    <operation name="opCancelOrderNotify">
        <input message="tns:msgCancelOrderNotifyRequest"/>
    </operation>
    <operation name="opGetOrder">
        <input message="tns:msgGetOrderRequest"/>
    </operation>
</portType>
21.2 Versioning Operation Definitions

The motivation behind this approach is to allow the operation MEP to be modified while avoiding an incompatible change. The primary limitation is that because the modified operation is effectively added as a new operation, it is required to have a different name.

```
<output message="tns:msgGetOrderResponse"/>
</operation>
</portType>
</definitions>
```

**Example 21.12**
The red text indicates a new `opCancelOrderNotify` operation that is added to the WSDL definition along with a new message definition. The bolded text represents the original `opCancelOrder`-related elements that are preserved.

NOTE
This technique is most commonly considered when following a Flexible versioning strategy. However, if you should need to follow this approach as part of a Strict versioning initiative, then it will force a new contract version the same way as if you were adding a new operation (as explained in the Forcing a New Contract Version when Adding an Operation (Strict) section).

Adding a Fault Message to an Existing Operation

As explained in Chapters 7 and 11, operations can be further assigned a pre-defined fault message that is transmitted by the Web service when certain exception conditions are encountered. The requirement for this change can occur when an operation in an established Web service contract did not previously need a fault message, but a new consumer wanting to use this operation introduces this requirement.

NOTE
You could argue that adding a fault message changes the MEP of the operation. However in WSDL 1.1, the four fundamental MEPs are not affected by the presence of the `fault` element.

*Adding a Fault Message (Strict, Flexible)*

This type of specific change may raise the question as to whether the addition of a fault message will affect the contract’s overall backwards compatibility. Technically it should
because we are making a significant change to both the contract and the behavior of an existing operation, as shown here:

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v3"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:tns="http://actioncon.com/contract/po/v3"
    xmlns:po="http://actioncon.com/schema/po"
    xmlns:ft="http://actioncon.com/schema/po/faults"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <documentation>Version 3.0</documentation>
  <types>
    <xsd:schema>
      <xsd:import namespace="http://actioncon.com/schema/po"
        schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
    </xsd:schema>
    <xsd:schema targetNamespace="http://actioncon.com/schema/po/faults">
      <xsd:element name="poFault" type="xsd:string"/>
    </xsd:schema>
  </types>
  <message name="msgSubmitOrderRequest">
    <part name="PurchaseOrder" element="po:purchaseOrder"/>
  </message>
  <message name="msgSubmitOrderResponse">
    <part name="Acknowledgement" element="po:acknowledgement"/>
  </message>
  <message name="msgSubmitOrderFault">
    <part name="POFault" element="ft:poFault"/>
  </message>
  <message name="msgCheckOrderRequest">
    <part name="PONumber" element="po:poNumber"/>
  </message>
  <message name="msgCheckOrderResponse">
    <part name="Status" element="po:status"/>
  </message>
  <message name="msgChangeOrderRequest">
    <part name="PurchaseOrder" element="po:purchaseOrder"/>
  </message>
  <message name="msgChangeOrderResponse">
    <part name="Acknowledgement" element="po:acknowledgement"/>
  </message>
  <message name="msgCancelOrderRequest">
    <part name="PONumber" element="po:poNumber"/>
  </message>
</definitions>
```
However, in some environments, the result of issuing a fault message to a consumer that isn’t expecting it is negligible. The fault message is transmitted, fails, and an error is recorded, but the consumer is never bothered by this event and therefore is not impacted by this change. Whether this is an option depends on the behavior of the runtime platform hosting both the Web service and the consumer.

Example 21.13

For this example, the contract designer decided to add the required type for the fault message definition by embedding a contract-specific schema into the <types> construct.
Although these circumstances may make it feasible to classify this change as compatible (and therefore only increase the minor contract version number), this is not a recommended approach. Future platform or other technology-related changes could alter the behavior of fault message processing, resulting in unpredictable results.

Extending the Contract with the Operation Containing the Fault Message (Flexible)

As with the change to operation MEPs, the alternative to simply adding a fault-capable version of the same operation definition also exists:

```xml
<portType name="ptPurchaseOrder">
  ...
  <operation name="opSubmitOrder">
    <input message="tns:msgSubmitOrderRequest"/>
    <output message="tns:msgSubmitOrderResponse"/>
  </operation>
  <operation name="opSubmitOrderWithFault">
    <input message="tns:msgSubmitOrderRequest"/>
    <output message="tns:msgSubmitOrderResponse"/>
    <fault message="tns:msgSubmitOrderFault"/>
  </operation>
  ...
</portType>
```

Example 21.14
The new `opSubmitOrderWithFault` operation containing the fault message is added to the WSDL definition.

Note that in this example, the `opSubmitOrder` and `opSubmitOrderWithFault` operations share the same message definitions for their input and output messages. If you intend to do this, be sure to read up on message dispatch issues that this can cause and the need to perhaps further supplement messages with hints as SOAP Action values (see Chapter 15 for more details).

**SUMMARY OF KEY POINTS**

- Some operation-level changes, such as adding a new operation, are backwards compatible.
- Most changes to operations, however, are incompatible unless alternative techniques are used, such as making a new change while preserving the original operation.
21.3 Versioning Port Type Definitions

This section explores how the versioning of a WSDL definition in general can affect the port type definition and also how a single contract can be evolved by acting as a container for multiple portType constructs (each representing a different version).

Adding a Version Identifier to the Port Type

To explicitly communicate the version of all operations within a given portType construct, you can embed the major version number in the portType element’s name attribute and update it every time the WSDL definition’s target namespace is updated.

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po/v3"
  xmlns="http://schemas.xmlsoap.org/wsdl/
  xmlns:tns="http://actioncon.com/contract/po/v3"
  xmlns:po="http://actioncon.com/schema/po"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <documentation>Version 3.0</documentation>
  ...
  <portType name="ptPurchaseOrder-v3">
    ...
  </portType>
  ...
</definitions>
```

Example 21.15
The portType name is modified to reflect the major version of the WSDL definition.

Versioning with Multiple Port Types (Flexible)

One alternative to creating a whole new WSDL definition document every time an incompatible change is applied to the abstract description is to create a new portType construct instead. Because a single WSDL definition can host multiple portType constructs, you can create a multi-interface WSDL document.

In the following example, the WSDL definition contains three portType constructs, each of which includes one of the incompatible changes described in the previous Versioning Operation Definitions section.

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po"
  xmlns="http://schemas.xmlsoap.org/wsdl/
  xmlns:po="http://actioncon.com/schema/po"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ...
</definitions>
```
Chapter 21: Versioning WSDL Definitions

```xml
<documentation>Versions 1.0, 2.0, 3.0</documentation>
<types>
  <xsd:schema>
    <xsd:import namespace="http://actioncon.com/schema/po"
                  schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
  </xsd:schema>
</types>
<message name="msgSubmitOrderRequest">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
<message name="msgSubmitOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgCheckOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>
<message name="msgCheckOrderResponse">
  <part name="Status" element="po:status"/>
</message>
<message name="msgChangeOrderRequest">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
<message name="msgChangeOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgCancelOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>
<message name="msgCancelOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>
<message name="msgGetOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>
<message name="msgGetOrderResponse">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>
<!-- For ptPurchaseOrder-v2 -->
<message name="msgSubmitOrdersRequest">
  <part name="PurchaseOrder" element="po:purchaseOrders"/>
</message>
```
21.3 Versioning Port Type Definitions

<message name="msgSubmitOrdersResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>

<portType name="ptPurchaseOrder-v1">
  <operation name="opSubmitOrder">
    <input message="tns:msgSubmitOrderRequest"/>
    <output message="tns:msgSubmitOrderResponse"/>
  </operation>
  <operation name="opCheckOrderStatus">
    <input message="tns:msgCheckOrderRequest"/>
    <output message="tns:msgCheckOrderResponse"/>
  </operation>
  <operation name="opChangeOrder">
    <input message="tns:msgChangeOrderRequest"/>
    <output message="tns:msgChangeOrderResponse"/>
  </operation>
  <operation name="opCancelOrder">
    <input message="tns:msgCancelOrderRequest"/>
    <output message="tns:msgCancelOrderResponse"/>
  </operation>
  <operation name="opGetOrder">
    <input message="tns:msgGetOrderRequest"/>
    <output message="tns:msgGetOrderResponse"/>
  </operation>
</portType>

<portType name="ptPurchaseOrder-v2">
  <operation name="opSubmitOrders">
    <input message="tns:msgSubmitOrdersRequest"/>
    <output message="tns:msgSubmitOrdersResponse"/>
  </operation>
  <operation name="opCheckOrderStatus">
    <input message="tns:msgCheckOrderRequest"/>
    <output message="tns:msgCheckOrderResponse"/>
  </operation>
  <operation name="opChangeOrder">
    <input message="tns:msgChangeOrderRequest"/>
    <output message="tns:msgChangeOrderResponse"/>
  </operation>
  <operation name="opCancelOrder">
    <input message="tns:msgCancelOrderRequest"/>
    <output message="tns:msgCancelOrderResponse"/>
  </operation>
  <operation name="opGetOrder">
    <input message="tns:msgGetOrderRequest"/>
  </operation>
</portType>
Chapter 21: Versioning WSDL Definitions

Because major versioning is represented by new port types, the target namespace of the definitions element remains unchanged from its original value and the major version numbers are incorporated into the portType element’s name attributes instead. The documentation element keeps track of supported versions by providing a list of version numbers.

Example 21.16
A single WSDL definition containing three portType constructs, each representing a different version of the abstract Web description.

```xml
<portType name="ptPurchaseOrder-v3">
  <operation name="opSubmitOrder">
    <input message="tns:msgSubmitOrderRequest"/>
    <output message="tns:msgSubmitOrderResponse"/>
  </operation>
  <!-- opCheckOrderStatus Removed 01/12 -->
  <operation name="opChangeOrder">
    <input message="tns:msgChangeOrderRequest"/>
    <output message="tns:msgChangeOrderResponse"/>
  </operation>
  <operation name="opCancelOrder">
    <input message="tns:msgCancelOrderRequest"/>
    <output message="tns:msgCancelOrderResponse"/>
  </operation>
  <operation name="opGetOrder">
    <input message="tns:msgGetOrderRequest"/>
    <output message="tns:msgGetOrderResponse"/>
  </operation>
</portType>
...
</definitions>
```

NOTE
Due to the fact that multiple portType constructs with different names exist, corresponding binding constructs will need to be defined. While providing multiple contract versions in a self-contained document, this approach can lead to extremely large-sized WSDL definitions that may become difficult to govern.
21.3 Versioning Port Type Definitions

Using Prefixes to Associate Versions

When adding new XML Schema types in support of different port type versions, namespace prefixes can be labeled to indicate version numbers.

The following example shows different version 2 and 3 `portType` constructs, each containing an operation that requires a new type that does not exist in the `purchaseOrder.xsd` schema:

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:tns="http://actioncon.com/contract/po"
    xmlns:po="http://actioncon.com/schema/po"
    xmlns:v2="http://actioncon.com/schema/po/errors"
    xmlns:v3="http://actioncon.com/schema/po/faults"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <documentation>Versions 1.0, 2.0, 3.0</documentation>
    <types>
        <xsd:schema>
            <xsd:import namespace="http://actioncon.com/schema/po"
                schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
        </xsd:schema>
        <xsd:schema targetNamespace="http://actioncon.com/schema/po/errors">
            <xsd:element name="statusError" type="xsd:string"/>
        </xsd:schema>
        <xsd:schema targetNamespace="http://actioncon.com/schema/po/faults">
            <xsd:element name="poFault" type="xsd:string"/>
        </xsd:schema>
    </types>
    <message name="msgSubmitOrderRequest">
        <part name="PurchaseOrder" element="po:purchaseOrder"/>
    </message>
    <message name="msgSubmitOrderResponse">
        <part name="Acknowledgement" element="po:acknowledgement"/>
    </message>
    <message name="msgSubmitOrderFault">
        <part name="POFault" element="v3:poFault"/>
    </message>
    <message name="msgCheckOrderRequest">
        <part name="PONumber" element="po:poNumber"/>
    </message>
</definitions>
```
Chapter 21: Versioning WSDL Definitions

```xml
<message name="msgCheckOrderResponse">
  <part name="Status" element="po:status"/>
</message>

<message name="msgCheckOrderResponseErr">
  <part name="StatusError" element="v2:statusError"/>
</message>

<message name="msgChangeOrderRequest">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>

<message name="msgChangeOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>

<message name="msgCancelOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgCancelOrderResponse">
  <part name="Acknowledgement" element="po:acknowledgement"/>
</message>

<message name="msgGetOrderRequest">
  <part name="PONumber" element="po:poNumber"/>
</message>

<message name="msgGetOrderResponse">
  <part name="PurchaseOrder" element="po:purchaseOrder"/>
</message>

<portType name="ptPurchaseOrder-v1">
  <operation name="opSubmitOrder">
    <input message="tns:msgSubmitOrderRequest"/>
    <output message="tns:msgSubmitOrderResponse"/>
  </operation>
  <operation name="opCheckOrderStatus">
    <input message="tns:msgCheckOrderRequest"/>
    <output message="tns:msgCheckOrderResponse"/>
  </operation>
  <operation name="opChangeOrder">
    <input message="tns:msgChangeOrderRequest"/>
    <output message="tns:msgChangeOrderResponse"/>
  </operation>
  <operation name="opCancelOrder">
    <input message="tns:msgCancelOrderRequest"/>
    <output message="tns:msgCancelOrderResponse"/>
  </operation>
  <operation name="opGetOrder">
    <input message="tns:msgGetOrderRequest"/>
    <output message="tns:msgGetOrderResponse"/>
  </operation>
</portType>
```
21.3 Versioning Port Type Definitions

```xml
<portType name="ptPurchaseOrder-v2">
  <operation name="opSubmitOrder">
    <input message="tns:msgSubmitOrderRequest"/>
    <output message="tns:msgSubmitOrderResponse"/>
  </operation>
  <operation name="opCheckOrderStatus">
    <input message="tns:msgCheckOrderRequest"/>
    <output message="tns:msgCheckOrderResponseErr"/>
  </operation>
  <operation name="opChangeOrder">
    <input message="tns:msgChangeOrderRequest"/>
    <output message="tns:msgChangeOrderResponse"/>
  </operation>
  <operation name="opCancelOrder">
    <input message="tns:msgCancelOrderRequest"/>
    <output message="tns:msgCancelOrderResponse"/>
  </operation>
  <operation name="opGetOrder">
    <input message="tns:msgGetOrderRequest"/>
    <output message="tns:msgGetOrderResponse"/>
  </operation>
</portType>

<portType name="ptPurchaseOrder-v3">
  <operation name="opSubmitOrder">
    <input message="tns:msgSubmitOrderRequest"/>
    <output message="tns:msgSubmitOrderResponse"/>
    <fault message="tns:msgSubmitOrderFault"/>
  </operation>
  <operation name="opCheckOrderStatus">
    <input message="tns:msgCheckOrderRequest"/>
    <output message="tns:msgCheckOrderResponse"/>
  </operation>
  <operation name="opChangeOrder">
    <input message="tns:msgChangeOrderRequest"/>
    <output message="tns:msgChangeOrderResponse"/>
  </operation>
  <operation name="opCancelOrder">
    <input message="tns:msgCancelOrderRequest"/>
    <output message="tns:msgCancelOrderResponse"/>
  </operation>
  <operation name="opGetOrder">
    <input message="tns:msgGetOrderRequest"/>
    <output message="tns:msgGetOrderResponse"/>
  </operation>
</portType>
```
Because a WSDL definition can contain multiple port types, you have the option of versioning multiple abstract descriptions within the same overall Web service contract.

When versioning multiple port types, prefixes can be labeled with version numbers to associate namespace values with parts of the WSDL definition related to different port type versions.

21.4 Versioning Concrete Descriptions

As mentioned earlier in this chapter, when making changes to operation and portType elements, you will always impact the concrete description because these elements are mirrored in the binding construct. However, in cases where there is more than one concrete description for a given abstract description, the binding and service constructs themselves can be versioned separately.

In Chapter 14 we discussed how the modularization features of the WSDL language allow one WSDL definition document to import another. The case study example in that chapter’s Modularization Mechanisms section explained how two different concrete descriptions in separate WSDL documents could be designed to import the same abstract description. If we imagine that one concrete description was released as a newer version of the other, then we introduce the need to add versioning identifiers.

Because the concrete descriptions exist in individual WSDL definitions, we have the opportunity to assign them separate target namespace values. Here we revisit this example by adding the appropriate version identifiers:

```xml
<definitions targetNamespace=
    "http://actioncon.com/contract/po/binding/v1"
    xmlns:tns=" http://actioncon.com/contract/po/binding/v1"
    xmlns:abs="http://actioncon.com/contract/po"
...
21.4 Versioning Concrete Descriptions

Example 21.18
Two separate WSDL definitions, each containing a different concrete description that imports the same abstract description.
In these examples, the name attribute of the binding, service, and port elements have also been modified to include the version number. This helps explicitly communicate the versioning identifier throughout the WSDL definition.

**SUMMARY OF KEY POINTS**

- Different versions of a WSDL concrete description can import the same abstract description.
- When following this approach, version numbers can be further embedded within the name of binding, part, and service elements.

### 21.5 WSDL Definitions and Forwards Compatibility

This chapter has intentionally focused on how changes to a WSDL definition tie into Strict and Flexible versioning approaches because options for supporting the Loose strategy are limited. The majority of changes we’ve explored so far have no direct bearing on enabling forwards compatibility and therefore support the Loose strategy only to the extent to which they affect backwards compatibility. However, some design options for enabling forwards compatibility do exist and are explained in these remaining sections.

**Decreasing Operation Granularity Levels**

The only way to really design a WSDL definition to accommodate a wider range of potential future consumers is to reduce its granularity levels so that they are more coarse and less fine.

An extreme example is simply collapsing finer-grained operations into a single coarse-grained operation, as shown here:

```xml
<definitions name="PurchaseOrder" targetNamespace="http://actioncon.com/contract/po"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:tns="http://actioncon.com/contract/po"
    xmlns:po="http://actioncon.com/schema/po">
    ...
    <portType name="ptPurchaseOrder">
        <operation name="opDoSomething">
            <input message="tns:msgRequest"/>
        </operation>
    </portType>
</definitions>
```
In this case, one operation is designed to accept a wide range of input message data. You could probably assume that the underlying message types would be using wildcards to accommodate such a design.

Alternatively, you could add the opDoSomething operation onto the portType construct displayed previously in Example 21.17 to accommodate future consumers that, for whatever reason, cannot use the existing, finer-grained operations. However, this type of design is not common.

### Using the WSDL 2.0 #any Attribute Value

Another example of building an extent of forwards compatibility into the WSDL definition is using the #any value provided by WSDL 2.0. As explained in Chapter 15, this special value allows an operation to be designed to receive (and respond with) any kind of message comprised of a valid XML document.

Here is an example of how the WSDL 2.0 version of the base Purchase Order Web service contract can be augmented in support of forwards compatibility:
Chapter 21: Versioning WSDL Definitions

This change is considered both backwards and forwards compatible because it does not impact existing consumers and will enable future consumers to send a broader range of input values to the two affected operations.

Example 21.21
The values for the `element` attributes of the input message for both the `opSubmitOrder` and `opChangeOrder` operations are changed from `po:PurchaseOrder` to `#any`.

This change is considered both backwards and forwards compatible because it does not impact existing consumers and will enable future consumers to send a broader range of input values to the two affected operations.
Provided next is another variation of the WSDL 2.0 base example where instead of decreasing the constraint granularity of existing operations, the contract is extended with a new, extremely coarse-grained operation that uses the #any attribute value for both input and output messages:

```xml
<description targetNamespace="http://actioncon.com/contract/po/v2"
  xmlns="http://schemas.xmlsoap.org/wsdl/
  xmlns:tns="http://actioncon.com/contract/po/v2"
  xmlns:po="http://actioncon.com/schema/po"
  xmlns:soapbind="http://www.w3.org/ns/wsdl/soap">
  <documentation>Version 2.2</documentation>
  <types>
    <xsd:schema>
      <xsd:import namespace="http://actioncon.com/schema/po"
        schemaLocation="http://actioncon.com/schema/purchaseOrder.xsd"/>
    </xsd:schema>
  </types>
  <interface name="ifPurchaseOrder">
    <operation name="opSubmitOrder"
      pattern="http://www.w3.org/2006/01/wsdl/in-out">
      <input messageLabel="In" element="po:purchaseOrder"/>
      <output messageLabel="Out" element="po:acknowlegement"/>
    </operation>
    <operation name="opCheckOrderStatus"
      pattern="http://www.w3.org/2006/01/wsdl/in-out">
      <input messageLabel="In" element="po:poNumber"/>
      <output messageLabel="Out" element="po:status"/>
    </operation>
    <operation name="opChangeOrder"
      pattern="http://www.w3.org/2006/01/wsdl/in-out">
      <input messageLabel="In" element="po:purchaseOrder"/>
      <output messageLabel="Out" element="po:acknowledgement"/>
    </operation>
    <operation name="opCancelOrder"
      pattern="http://www.w3.org/2006/01/wsdl/in-out">
      <input messageLabel="In" element="po:poNumber"/>
      <output messageLabel="Out" element="po:acknowledgement"/>
    </operation>
    <operation name="opProcessOrder"
      pattern="http://www.w3.org/2006/01/wsdl/in-out">
      <input messageLabel="In" element="#any"/>
    </operation>
  </interface>
</description>
```
This type of operation can provide a broad “catch all” allowing practically any range of input data and also being able to respond with just about any form of output message.

**SUMMARY OF KEY POINTS**

- There aren’t many ways to support forwards compatibility using the WSDL language.
- One option is to use the `#any` attribute and another approach is to simply reduce operation granularity levels.
Chapter 22

Versioning Message Schemas

22.1 Basic Terms and Concepts
22.2 XML Schema and WSDL Target Namespaces
22.3 Strict Versioning
22.4 Flexible Schema Versioning (Using Optional Components)
22.5 Loose Schema Versioning (Using Wildcards)
No part of the Web service contract demands as much versioning-related attention as the XML Schema definitions that form the basis of input, output, and fault messages. Introducing changes to existing type definitions and data structures used by WSDL message definitions can be relatively simple and uneventful, or it can produce a ripple effect across consumers that (especially with agnostic services) can impose significant impact.

In the upcoming sections, we'll be exploring the following three common change types:

- adding a new schema component
- removing an existing schema component
- renaming an existing schema component
- modifying the constraint of an existing schema component

This chapter contains three sections that correspond to the Flexible, Loose, and Strict versioning strategies we introduced in Chapter 20. Each one of these sections explains how these change types can be carried out in accordance with the rules of the versioning strategy.

NOTE

Because some of the versioning practices in this chapter incorporate the use of XML Schema wildcards, you might want to revisit the Wildcards section in Chapter 12 before proceeding.

22.1 Basic Terms and Concepts

Re-Introducing the “Component”

Back in Chapter 6 we briefly introduced the term “component” as it is commonly referenced when discussing XML and XML Schema. Essentially, a component, from an XML Schema perspective, represents a fundamental part of the schema and is commonly used to refer to an attribute or element. So far in this book we haven’t really used this term in order to avoid confusing XML Schema components with component-based programs used by most Web services to host the underlying service logic.
22.1 Basic Terms and Concepts

However, given that we’re focused solely on contract versioning from hereon, and because several versioning techniques apply to both elements and attributes, we’re bringing the term back. You will sometimes see this term further qualified as *schema component*.

**Content Sets**

The following new terms allow us to make a distinction between the parts of a message schema that do and do not provide formal component declarations:

- **Defined Set** – This represents the part of a message definition that has been predefined via declarations in its underlying schema. The defined set corresponds to the minimum constraints and requirements that messages must comply with in order to be considered valid.

- **Allowed Set** – The maximum range of what a message can contain to remain compatible with the message definition’s type. The allowed set encompasses the defined set. If the schema does not use wildcards, the allowed set is equivalent to the defined set. If wildcards are used, then the allowed set will exceed the defined set to whatever extent the wildcards allow.

- **Unknown Set** – This simply represents the gap (if any) between the defined set and the allowed set of a given message definition. It refers to the range of unknown (not pre-defined) data that a message can contain while still remaining valid.

- **Recognized Set** – When message recipients are designed to accommodate a measure of content that is part of the unknown set, we end up with another classification that corresponds to the total amount of content recognized. The recognized set always encompasses the defined set and may encompass some or all of the unknown set. We don’t actually begin discussing the recognized set until the *Working with Unknown Content* section in Chapter 23, where the distinction between recognized and allowed content becomes relevant in relation to determining processing options for content that is part of the allowed set but not part of the recognized set.

Let’s take a look at our example to determine how these terms apply:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po"
    xmlns="http://actioncon.com/schema/po">
    <xsd:element name="LineItem" type="LineItemType"/>
</xsd:schema>
```
All of the code highlighted in red belongs to the defined set because it establishes pre-defined structure and declarations. The bolded text represents the part of the schema classified as the unknown set because it allows a wide assortment of undefined (unknown) data to be accepted. Finally, both the red and the bolded text is considered the allowed set because the combination of the defined and unknown sets represent the range of allowable message content.

Example 22.1
A sample complex type with wildcards helps us sort out the difference between a defined set and an unknown set, which together comprise the overall allowed set.

All of the code highlighted in red belongs to the defined set because it establishes pre-defined structure and declarations. The bolded text represents the part of the schema classified as the unknown set because it allows a wide assortment of undefined (unknown) data to be accepted. Finally, both the red and the bolded text is considered the allowed set because the combination of the defined and unknown sets represent the range of allowable message content.

NOTE
The recognized set was not represented in the preceding example because it is dependent on the service implementation.

Versioning and the UPA Rule
In Chapter 12 we briefly introduced the Unique Particle Attribution (UPA) rule that is part of the XML Schema specification. This rule dictates that every element must be attributed to exactly one “particle” (or construct) in a schema.

What this refers to specifically is the use of optional elements together with wildcard elements. You cannot design a schema in which an optional element is followed by a wildcard element that allows the same namespace as the optional element. This is because by having an optional element and a wildcard in the same namespace, you essentially establish two parts of the same complex type that would validate the same element value.
22.1 Basic Terms and Concepts

This rule primarily exists to improve the efficiency of parsers. Without the UPA rule, a parser might need to “look ahead” in a type definition to determine if an optional element declaration exists for a given element value or whether the value should be associated with the wildcard. Having to do this for every possible wildcard value can burn up a lot of processing cycles. The UPA rule essentially prevents the need for such a look ahead.

An Example of a UPA Violation

In the following example, there is a conflict because the po2:available element instance is valid under both the optional po2:available element and the ##other wildcard. The possibility of the element matching two or more “particles” violates the unique part of the UPA rule.

```
xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://actioncon.com/schema/po"
xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element ref="po2:available" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

*Example 22.2*

An illegal schema due to the fact that it violates the UPA rule.

The UPA rule is relevant to schema versioning because of the restriction it places on the use of optional elements and wildcards, both of which represent fundamental tools to enable backwards and forwards compatibility.
662

Chapter 22: Versioning Message Schemas

Base Example

Each of the upcoming versioning techniques will be demonstrated on the same base schema sample:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po"
    xmlns="http://actioncon.com/schema/po"
    elementFormDefault="qualified"
    version="1.0">
    <xsd:element name="LineItem" type="LineItemType"/>
    <xsd:complexType name="LineItemType">
        <xsd:sequence>
            <xsd:element name="productID" type="xsd:string"/>
            <xsd:element name="productName" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Example 22.3

The `LineItem` element comprised of the `LineItemType` complex type representing the ID and name of a product.

This code was borrowed from the Purchase Order service schema first described in Chapter 6, and then simplified for use in these versioning chapters. It shows a relatively basic complex type comprised of two elements with simple content.

As part of a SOAP message, an instance of this schema might look like this:

```xml
<LineItem xmlns="http://actioncon.com/schema/po">
  <productID>AY2345</productID>
  <productName>Service Blaster 2000</productName>
</LineItem>
```

Example 22.4

An XML document instance of the base schema.

In the upcoming versioning scenarios, the `LineItemType` complex type is extended with an available element that represents a Boolean property of a game product that indicates whether the product is currently available.
22.1 Basic Terms and Concepts

Versioning Conventions for Examples

Based on the version number provided in the `version` attribute and the fact that there is no version identifier in the namespace, you can assume that this base schema is version “1.0” in relation to any version numbers you will see hereafter.

As with the convention we used with the WSDL examples in Chapter 21, major versions will be represented in the schema target namespace. With the WSDL definitions we kept track of minor version increments using the WSDL documentation element. We could actually do the same with XML Schema by adding the `xsd:annotation` and `xsd:documentation` elements, as shown here:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po"
    xmlns="http://actioncon.com/schema/po">
    <xsd:annotation>
        <xsd:documentation>
            Version 1.0
        </xsd:documentation>
    </xsd:annotation>
    ...
</xsd:schema>
```

**Example 22.5**

Major and minor version numbers presented within the `xsd:documentation` construct.

However, given how bulky this makes our schemas, we’re opting for the version attribute of the schema element itself, as follows:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po"
    xmlns="http://actioncon.com/schema/po"
    version="1.0">
    ...
</xsd:schema>
```

**Example 22.6**

The `xsd:schema` element’s `version` attribute used to express our major and minor version numbers.
SUMMARY OF KEY POINTS

- A schema is comprised of content sets that represent defined, allowed, unknown, and recognized content.
- Defined content plus unknown content is equal to allowed content. Recognized content is specific to the service implementation.
- UPA rule violation is a constant consideration when designing schemas for forwards compatibility.
- This section establishes a base schema that will be used as the basis for upcoming examples.

22.2 XML Schema and WSDL Target Namespaces

In this chapter we’ll be repeatedly discussing how major version changes result in a change to the XML Schema definition’s target namespace value. This approach is based on common conventions we use in our examples and does not preclude other methods that do not involve namespaces at all. However, because we use this approach, we need to establish how the creation of new XML Schema target namespace values relates to the types of changes that we covered in the previous chapter, which result in the need for new WSDL target namespaces.

The rule of thumb is quite straightforward. If an XML Schema definition undergoes a change that requires a new target namespace for that schema, that change will propagate to the WSDL level, resulting in a new target namespace for the WSDL definition as well.

The best way to understand this relationship is to simply view the XML Schema content as an extension to the WSDL document. We can just as easily embed the XML Schema types and element declarations into the WSDL types construct as we can import them. Therefore, their physical location has no bearing on the fact that these definitions collectively represent the final Web service contract.

Note that one consideration related to XML Schema definitions that exist as separate files is when those schemas are shared (as per the Schema Centralization pattern). This is explored further in Chapter 23.

NOTE

Minor version number changes in XML Schema definitions can optionally also be reflected in the corresponding WSDL definition in order to communicate the change in both documents.
22.3 Strict Versioning

The rules of the Strict versioning approach are very simple: Any type of change forces a new contract version. Therefore, the different code examples in the following sections have pretty much the same end result.

**Adding a New Schema Component**

Here we can see a sample version 2 of the base schema. The `available` element declaration was added to the `LineItemType` complex type, requiring that the namespace value be changed.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po/v2"
    xmlns="http://actioncon.com/schema/po/v2"
    version="2.0">
    <xsd:element name="LineItem" type="LineItemType"/>
    <xsd:complexType name="LineItemType">
        <xsd:sequence>
            <xsd:element name="productID" type="xsd:string"/>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="available" type="xsd:boolean"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Example 22.7
The `available` element is added, forcing a change in the namespace value.

In the preceding example the addition of the `available` element introduced an incompatible change because the element was required.

As per the rules of this versioning approach, even a compatible change would require a new namespace value, as shown here:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po/v2"
    xmlns="http://actioncon.com/schema/po/v2"
    version="2.0">
    <xsd:element name="LineItem" type="LineItemType"/>
    <xsd:complexType name="LineItemType">
        <xsd:sequence>

```

Web Service Contract Design and Versioning for SOA-- 013613517X
Removing an Existing Schema Component

Deleting a component from an existing message schema naturally results in an incompatible change that requires the issuance of a new target namespace value, as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po/v2"
    xmlns="http://actioncon.com/schema/po/v2"
    version="2.0">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <!-- productName Removed 09/12 -->
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 22.9

The productName element declaration is deleted, resulting in a new major version of the schema.

In this case, the removal of the element declaration is logged via a human-readable comment for future reference purposes.

Renaming an Existing Schema Component

Renaming an established element will have the same impact as any other change, as follows:

```
<xsd:element name="available" type="xsd:boolean"
    minOccurs="0"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Example 22.8

The available element is added with the minOccurs attribute set to "0," making this an optional element, but still requiring a change to the namespace.
22.3 Strict Versioning

Furthermore, because you do not need to rely on the use of optional elements or wildcards, message exchanges are always predictable and validated to whatever extent necessary at the contract level.

Modifying the Constraint of an Existing Schema Component

Due to the rules of the Strict approach, changes that result in either an increase or decrease of a component’s constraint granularity will force a new version of the schema.

For example, the following changes make each of the previously required elements optional:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po/v2"
    xmlns="http://actioncon.com/schema/po/v2"
    version="2.0">
    <xsd:element name="LineItem" type="LineItemType"/>
    <xsd:complexType name="LineItemType">
        <xsd:sequence>
            <xsd:element name="productID" type="xsd:string" minOccurs="0"/>
            <xsd:element name="productName" type="xsd:string" minOccurs="0"/>
            <xsd:element name="available" type="xsd:boolean" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Example 22.10

The `productName` element is renamed to `productName2`, forcing another change to the namespace value.

Furthermore, because you do not need to rely on the use of optional elements or wildcards, message exchanges are always predictable and validated to whatever extent necessary at the contract level.
Even though these are all considered compatible changes, a new schema version is still required, as indicated in the target namespace. Similarly, making an incompatible change (such as making an optional element required) will have the same result.

### SUMMARY OF KEY POINTS

- The Strict versioning approach requires a new target namespace in response to any change to the message schema content.
- The end result of applying this approach is pretty much the same for most types of changes, regardless of whether they are compatible or incompatible changes.

### 22.4 Flexible Schema Versioning (Using Optional Components)

When applying the Flexible strategy to schema versioning, we look for ways to leverage features of the XML Schema language that can help us implement different types of changes while continuing to maintain backwards compatibility. The following set of examples explores a range of techniques for accommodating both compatible and incompatible changes.

**Adding a New Schema Component**

The addition of a new component to an existing schema is considered an incompatible change if the new component is required, and a compatible change if it’s optional.

As with the Strict approach, the addition of a new required element declaration forces a new major schema version:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po/v2"
    xmlns="http://actioncon.com/schema/po/v2"
>
```

---

Example 22.11

The minOccurs attribute set to “0” on all elements within the LineItemType complex type.
In order to accomplish consistent backwards compatibility, the Flexible versioning strategy encourages us to make any new element or attribute declarations optional. In the following example, we extend the base schema with an optional `available` element and also indicate the change by incrementing the minor version number value in the `version` attribute:

```xml
<xs:element name="LineItem" type="LineItemType"/>
<xs:complexType name="LineItemType">
   <xs:sequence>
      <xs:element name="productID" type="xsd:string"/>
      <xs:element name="productName" type="xsd:string"/>
      <xs:element name="available" type="xsd:boolean"
                  minOccurs="1"/>
   </xs:sequence>
</xs:complexType>
</xs:schema>
```

**Example 22.12**
The highlighted declaration is added making the `available` element a required part of the `LineItemType` complex type.

```xml
<xs:element name="LineItem" type="LineItemType"/>
<xs:complexType name="LineItemType">
   <xs:sequence>
      <xs:element name="productID" type="xsd:string"/>
      <xs:element name="productName" type="xsd:string"/>
      <xs:element name="available" type="xsd:boolean"
                  minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
</xs:schema>
```

**Example 22.13**
The `available` element is added with the `minOccurs` attribute set to "0," making the existence of the element in message instances optional.
The addition of a new schema component now allows messages to optionally include the available element, as follows:

```xml
<LineItem xmlns="http://actioncon.com/schema/po">
   <productID>AY2345</productID>
   <productName>Service Blaster 2000</productName>
   <available>true</available>
</LineItem>
```

**Example 22.14**
An instance of a message document that contains the new available element.

### Placing New Component Declarations into Separate Schemas

Note that the new available element could have been declared in a separate schema that can be referenced from the base schema, as shown here:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://actioncon.com/schema/po"
            version="1.1">
   <xsd:element name="LineItem" type="LineItemType"/>
   <xsd:complexType name="LineItemType">
      <xsd:sequence>
         <xsd:element name="productID" type="xsd:string"/>
         <xsd:element name="productName" type="xsd:string"/>
         <xsd:element ref="available" minOccurs="0"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

**Example 22.15**
The LineItemType complex type in the first schema contains an element that references another in a separate schema.

This approach may be preferable when new elements need to be governed by separate custodians, or if the new elements need to be placed into a different namespace as follows:
22.4 Flexible Schema Versioning (Using Optional Components)

Removing an Existing Schema Component

The removal of a component declaration from an existing XML Schema definition results in an incompatible change that forces a new schema and WSDL definition version. Therefore, when carrying out the Flexible versioning strategy, this change has the same results as with the Strict approach, as explained earlier in the corresponding Removing an Existing Schema Component part of the previous Strict Versioning section.

Renaming an Existing Schema Component

By default, changing the name of an existing component will result in an incompatible change that requires a new schema target namespace and a new corresponding contract version:

Example 22.16
The externally referenced element is placed into a separate namespace.
This type of change would make the following message instance invalid:

```xml
<LineItem xmlns="http://actioncon.com/schema/po">
  <productID>AY2345</productID>
  <productName2>Service Blaster 2000</productName2>
</LineItem>
```

**Example 22.18**
An instance of a message document in which an element name has been changed.

However, with the Flexible strategy, schema components can be renamed as part of a compatible change by using the `xsd:choice` group element to preserve the old element name alongside the new one:

```xml
<xsd:complexType name="LineItemType">
  <xsd:sequence>
    <xsd:element name="productID" type="xsd:string"/>
    <xsd:choice>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="productName2" type="xsd:string"/>
    </xsd:choice>
    <xsd:element name="available" type="xsd:boolean"/>
  </xsd:sequence>
</xsd:complexType>
```

**Example 22.19**
The `productName` and `productName2` element declarations are wrapped in a choice group.
22.4 Flexible Schema Versioning (Using Optional Components)

Modifying the Constraint of an Existing Schema Component

Adjusting the validation rules for a given component is a common type of change, especially when the validation logic is tied to business policies or rules. In the following example, the data type for the `productID` element has been changed to `xsd:integer`, and the `maxOccurs` attribute for the `productName` element has been set to “unbounded,” allowing multiple occurrences of the element to exist:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po/v2"
    xmlns="http://actioncon.com/schema/po/v2"
    elementFormDefault="qualified"
    version="2.0">
    <xsd:element name="LineItem" type="LineItemType"/>
    <xsd:complexType name="LineItemType">
        <xsd:sequence>
            <xsd:element name="productID" type="xsd:integer"/>
            <xsd:element name="productName" type="xsd:string"
                maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Example 22.20

The `type` attribute of the `productID` element declaration is changed from `xsd:string` to `xsd:integer`, and the `maxOccurs` attribute of the `productName` element declaration is changed to a value of “unbounded.”

The change made to the `productID` element is incompatible because the `xsd:integer` data type is more restrictive than the original `xsd:string` type, whereas the addition of `maxOccurs="unbounded"` is a compatible change that will not impact existing consumers.

As a result, the following message sent to the new schema will fail validation:

```xml
<LineItem xmlns="http://actioncon.com/schema/po">
    <productID>AY2345</productID>
    <productName>Service Blaster 2000</productName>
    <productName>Service Blaster 2010</productName>
</LineItem>
```

Example 22.21

A message instance that does not comply with the preceding schema.
The multiple productName elements are legal, but the value of the productID element is no longer valid, since it does not comply with the xsd:integer type.

Similarly, some or all of the previously existing schema types can be designed with increased flexibility by also becoming optional and by increasing their allowable occurrences. In the following example, we decide that the only required element should be productID, and therefore also make the productName element optional:

```xml
<xsd:complexType name="LineItemType">
  <xsd:sequence>
    <xsd:element name="productID" type="xsd:string"/>
    <xsd:element name="productName" type="xsd:string" minOccurs="0"/>
    <xsd:element name="available" type="xsd:boolean" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

**Example 22.22**
The productName element is made optional by setting its minOccurs attribute to "0."

We might then discover that game products can be renamed several times, but that they are required to retain old names together with any new names. In this case we need to allow the productName element to occur more than just once (which is the default). Therefore, we add the maxOccurs attribute to the element with a setting of "unbounded," as shown here:

```xml
<xsd:complexType name="LineItemType">
  <xsd:sequence>
    <xsd:element name="productID" type="xsd:string"/>
    <xsd:element name="productName" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="available" type="xsd:boolean" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

**Example 22.23**
The productName element has the maxOccurs attribute set to "unbounded," allowing this element to repeat.
We could have made these changes to the `productName` element in advance, but we also had the option of adding these attribute values subsequent to the initial deployment of the Purchase Order service, because they retained backwards compatibility.

Earlier we demonstrated the use of the `xsd:choice` group to accommodate component name changes. Note that due to restrictions in the XML Schema language, this group cannot help us overcome changes to schema component constraints in the same way.

For example, the following is not allowed:

```xml
<xsd:complexType name="LineItemType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productID" type="xsd:decimal"/>
    </xsd:choice>
    <xsd:element name="productName" type="xsd:string"/>
    <xsd:element name="available" type="xsd:boolean"/>
  </xsd:sequence>
</xsd:complexType>
```

Example 22.24
An invalid choice group comprised of two element declarations that only differ in the values of their type attributes.

**NOTE**
Overuse of `minOccur="0"` and `maxOccur="unbounded"` can result in vague Web service contracts with reduced capacity to validate incoming and outgoing data. This shifts the responsibility of performing the actual validation into the service logic, further requiring the service to contain exception handling routines for responding to invalid messages.

**SUMMARY OF KEY POINTS**

- When following the Flexible versioning strategy, most schema changes can be backwards-compatible via the use of the `minOccur` attribute.
- Incompatible changes, such as renaming or removing existing XML Schema components, will likely force a new major version unless special techniques are used.
22.5 Loose Schema Versioning (Using Wildcards)

The added goal of supporting forwards compatibility within our schema designs makes this the most challenging approach to follow. The following sections demonstrate how various extents of forwards compatibility can be achieved through the use of XML Schema wildcards.

Adding a New Schema Component

An alternative approach to managing message type changes is with the use of wildcards. Because a wildcard will accept a wide range of elements, this approach has the potential to enable Web service contracts to be designed in support of both backwards and forwards compatibility.

NOTE
As explained in Chapter 12, additional elements that are added and then validated against wildcards are referred to as replacement elements.

In the following example, we’ve extended our base XML schema to add the `xsd: any` and `xsd: anyAttribute` wildcard declarations:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po"
    xmlns="http://actioncon.com/schema/po"
    elementFormDefault="qualified"
    version="1.0">
    <xsd:element name="LineItem" type="LineItemType"/>
    <xsd:complexType name="LineItemType">
        <xsd:sequence>
            <xsd:element name="productID" type="xsd:string"/>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:any namespace="##any" processContents="lax"
                minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:anyAttribute namespace="##any"/>
    </xsd:complexType>
</xsd:schema>
```

Example 22.25

The base schema is extended with `xsd: any` and `xsd: anyAttribute` wildcards. Our assumption is that we designed these extensions into version 1.0 of our base schema in support of future Loose versioning.
22.5 Loose Schema Versioning (Using Wildcards)

In this case, the wildcards are added to the end of the LineItemType construct, which means that all changes and extensions to that type will need to be implemented via wildcards. As a result, the initial defined set (comprised of the productID and productName element declarations) will never be allowed to grow.

For the previously displayed schema, the following message would still be valid because it simply complies with the types in the defined set:

```
<LineItem xmlns="http://actioncon.com/schema/po">
  <productID>AY2345</productID>
  <productName>Service Blaster 2000</productName>
</LineItem>
```

Example 22.26
A basic XML document instance that conforms to the original element declarations.

And, the same message extended with a previously undeclared element would also be considered valid:

```
<LineItem xmlns="http://actioncon.com/schema/po">
  <productID>AY2345</productID>
  <productName>Service Blaster 2000</productName>
  <available>true</available>
</LineItem>
```

Example 22.27
An extended XML document instance that conforms to the schema because of its use of wildcards.

**Using Wrapper Elements for Wildcards**

An alternative design to the previously displayed schema is to wrap the wildcard into a separate optional element, as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po"
  version="1.1">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Web Service Contract Design and Versioning for SOA-- 013613517X
This approach is desirable when you want to isolate replacement elements into a separate construct within the message document. For example, the previous message would now need to be structured like this:

```xml
<LineItem xmlns="http://actioncon.com/schema/po"
  <productID>AY2345</productID>
  <productName>Service Blaster 2000</productName>
  <extension>
    <available>true</available>
  </extension>
</LineItem>
```

**Example 22.28**
The `xsd:any` and `xsd:anyAttribute` wildcards wrapped in a separate `extension` element declaration.

**Example 22.29**
A message instance containing the required extension wrapper element in order to conform to the preceding schema.

---

**Schemas for Replacement Elements**

Example 22.15 in the *Flexible Schema Versioning* section demonstrated how a new element can be declared separately in its own schema and then referenced from the base schema via the use of the `ref` attribute of the `element` element.

When working with wildcards, in either of the schema designs we just explored (with and without wrapper elements), the `available` element can also be declared in a separate schema definition, as shown here:

```xml
<xsd:element name="productName" type="xsd:string"/>
<xsd:element name="extension" type="ExtensionType"
  minOccurs="0"/>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="ExtensionType">
  <xsd:sequence>
    <xsd:any namespace="##targetNamespace"
      processContents="lax" minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:anyAttribute namespace="##any"/>
</xsd:complexType>
</xsd:schema>
```

**Example 22.28**
The `xsd:anyAttribute` and `xsd:any` wildcards wrapped in a separate `ExtensionType` declaration.
You will notice the absence of a `ref` attribute in the base schema. This attribute is not supported with wildcard elements. Several processors are smart enough to associate the two schemas automatically at runtime by using the replacement element name as the primary point of reference. However, you should confirm this support before basing your contracts on this design. Alternatively, you can try to use the standard XML Schema import or include statements to either join one schema with another or have them both pulled into a WSDL definition.

Note that in the previous example, the allowed target namespace values for the replacement element schema are determined by the `namespace` attribute setting of the wildcard element, which also provides the `processContents` attribute that enables us to set whether the schema will actually be used for validation processing. Both of these attributes are explained in detail in the upcoming *Modifying the Constraint of an Existing Schema Component* section.
Removing an Existing Schema Component

If you run into the requirement to remove a component declaration from an established defined set of a Web service contract, then there are several options you can explore in order to avoid turning this into an incompatible change. When working with wildcards, these options relate to the vicinity of the component declaration to the wildcard, as explained in the following two sections.

Removing an Element Preceding a Wildcard

In this example, we need to remove the declaration for the `productName` element:

```xml
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po"
    xmlns="http://actioncon.com/schema/po"
    version="1.0">
  <xsd:element
    name="LineItem"
    type="LineItemType"/>
  <xsd:complexType
    name="LineItemType">
    <xsd:sequence>
      <xsd:element
        name="productID"
        type="xsd:string"
        minOccurs="0"/>
      <xsd:element
        name="productName"
        type="xsd:string"/>
      <xsd:any
        namespace="##any"
        processContents="lax"
        minOccurs="0"
        maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:anyAttribute
      namespace="##other"/>
  </xsd:complexType>
</xsd:schema>
```

**Example 22.31**
The `productName` element declaration has been identified as an XML Schema component that needs to be removed.
22.5 Loose Schema Versioning (Using Wildcards)

Because the element declaration for `productName` is immediately followed by the wildcard, we can simply delete the highlighted line of text, as shown here:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://actioncon.com/schema/po"
xmns="http://actioncon.com/schema/po"
version="1.1">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"
        minOccurs="0"/>
      <!-- productName Removed 09/12 -->
      <xsd:any namespace="##any" processContents="lax"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:anyAttribute namespace="##other"/>
  </xsd:complexType>
</xsd:schema>
```

**Example 22.32**
The `LineItemType` construct after the `productName` element declaration has been removed.

The removal of the `productName` element declaration was a simple compatible change because the wildcard can now continue to accept the `productName` element in messages from existing consumers, while allowing new consumers to no longer have to provide this formerly required element.

*Removing an Element Not Preceding a Wildcard*

It is more difficult to remove an element declaration that does not directly precede a wildcard declaration. In the following example, we want to remove the `productID` element, which is located at the beginning of the `sequence` construct and precedes the `productName` element declaration:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://actioncon.com/schema/po"
xmns="http://actioncon.com/schema/po"
version="1.0">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
```

Web Service Contract Design and Versioning for SOA-- 013613517X
In this case, simply replacing the element with a wildcard (as per the previous section) would result in a UPA rule violation.

As an alternative, we can preserve the productID element declaration and simply “relax” its constraints by changing its type to xsd:anyType and adding minOccurs="0", as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po"
    xmlns="http://actioncon.com/schema/po"
    version="1.1">
    <xsd:element name="LineItem" type="LineItemType"/>
    <xsd:complexType name="LineItemType">
        <xsd:sequence>
            <xsd:element name="productID" type="xsd:anyType"
                minOccurs="0"/>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:any namespace="##any" processContents="lax"
                minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:anyAttribute namespace="##other"/>
    </xsd:complexType>
</xsd:schema>
```

**Example 22.34**
The value of the type attribute of the productID element is changed to "xsd:anyType."

While this technique does not actually physically remove the productID element declaration, it decreases its constraint granularity to a point where it resembles a wildcard and it further limits the impact to a compatible change.
### Renaming an Existing Schema Component

When having to rename an established component within a schema, the techniques covered in the corresponding *Renaming an Existing Schema Component* part of the *Flexible Schema Versioning* section can be applied.

### Modifying the Constraint of an Existing Schema Component

The examples we covered in the corresponding *Modifying the Constraint of an Existing Schema Component* part of the *Flexible Schema Versioning* section also apply to the Loose versioning approach in how they support backwards compatibility for the defined set. Beyond existing component declarations, there are additional considerations related to the constraint granularity of wildcard declarations based on the special attributes used with XML Schema wildcards.

Let’s revisit the `xsd:any` wildcard to study its attribute settings:

```xml
<xsd:any namespace="##any" processContents="lax"
    minOccurs="0" maxOccurs="unbounded"/>
```

The purpose of the `minOccurs` and `maxOccurs` attribute settings is pretty straightforward in that they clearly indicate that additional elements beyond the defined set are optional and can be added as many times as you want.

What’s of particular interest to use in relation to Loose versioning are the `namespace` and `processContents` attributes that we first introduced back in Chapter 12. The following two sections recap the allowed values for these attributes and discuss how they relate to versioning options.

#### The namespace Attribute

The `namespace` attribute determines the namespaces that replacement elements can belong to, as follows:

- `##any` – (the default value) any namespace or no namespace
- `##other` – any namespace other than the target namespace

---

**NOTE**

This may, in fact, not be the best example for the use of the `xsd:anyType` type because most of what’s allowed by `xsd:anyType` is also allowed by the `xsd:string` data type. A more effective scenario would be one where the original `productId` element declaration contained more restrictive constraints.
• ##targetNamespace – the target namespace of the schema document
• #local – no namespace
• one or more specific namespaces, separated by whitespace

If the replacement element is associated with a namespace other than what is allowed by this attribute, the validation will fail. Therefore, we need to take a closer look to explore the impact of these values on a versioning strategy.

The namespace="##any" setting is clearly the most flexible option in that the replacement elements can belong to any namespace. Although it provides the greatest range of replacement elements that can be accepted by a given service contract, it also increases the chances of generating UPA violations if the last part of the defined set is optional.

The use of the namespace="##other" setting forces all replacement elements to be associated with a namespace that is different from the existing schema’s target namespace. This can introduce more design effort to ensure that new namespace values are always used for replacement elements, but it also guarantees that the UPA rule is consistently adhered to.

If you are employing a versioning approach that is part of a tightly controlled governance program, then you can alternatively use the option whereby the namespace attribute is populated with a specific list of allowed namespaces. This way, messages must always comply with namespaces that are issued by a custodian, and the namespace="##targetNamespace" and namespace="##local" settings can be used to support specific requirements and conventions that are part of a controlled governance effort.

To understand these attribute settings better, let’s take a look at some examples. In the following scenario, the original schema has a wildcard element with a namespace value of “##other” and a possible replacement element is separately declared in a schema that shares the same target namespace as the base schema:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po"
    xmlns="http://actioncon.com/schema/po"
    version="1.1">
  <xsd:element name="LineItem" type="LineItemType"/>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```
22.5 Loose Schema Versioning (Using Wildcards)

This example would raise an error because the use of the `namespace="##other"` setting requires that replacement elements exist in a separate namespace. For this to work, either the namespace attribute of the wildcard needs to be changed to a value of "##any" or "##targetNamespace" or it needs to be populated with the actual namespace value.

Here’s a correct example of how the schema that declares the available element can work with the primary schema from Example 22.35:

```
<xs:complexType>
  <xs:sequence>
    <xs:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
    <xs:anyAttribute namespace="##other"/>
  </xs:complexType>
</xs:schema>
```

Example 22.35

The `##other` attribute value used to require that elements belonging to the unknown set reside in a namespace other than the target namespace of the primary schema document. However, the second schema document is still using the same target namespace.

The `processContents` Attribute

The setting of this attribute allows you to determine the extent to which replacement elements will be validated, as follows:
• **strict** – (default) Elements must be declared and validated. If they are invalid or no declarations are found, an error is raised.

• **lax** – The processor will look for declarations for those elements (based on their names) and if it finds them, it will validate them (if it doesn’t find them, it will not raise an error).

• **skip** – No validation will be performed on the replacement elements.

A value of “strict” requires that an element declaration exists for any replacement element. In this case, separate schemas will likely need to be created with namespace values that comply with the namespace attribute setting. When wildcards are in frequent use by a range of different consumer programs, it is not uncommon for the same base schema that establishes the defined set to be potentially extended with a variety of different schemas.

Similarly, the “lax” setting will use existing schemas to validate replacement elements. However, the absence of any required schemas will not cause replacement elements to be rejected.

<table>
<thead>
<tr>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>With both the “strict” and “lax” settings, the processor will usually attempt to locate the appropriate schemas based on the replacement element name values.</td>
</tr>
</tbody>
</table>

The value of “skip” will simply tell the processor to allow any replacement elements that comply with the namespace attribute setting to pass through to the core service logic.

As an example, the following schema definition for the available replacement element will be ignored:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://actioncon.com/schema/po"
    xmlns="http://actioncon.com/schema/po"
    version="1.1">
    <xsd:element name="LineItem" type="LineItemType"/>
    <xsd:complexType name="LineItemType">
        <xsd:sequence>
            <xsd:element name="productID" type="xsd:string"/>
            <xsd:element name="productName" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```
22.5 Loose Schema Versioning (Using Wildcards)

How these attribute values affect your versioning approach depends on how much control you need to exercise over how replacement elements are defined. It is often advisable to begin with a setting of “strict” and to then only reduce it to “lax” or even “skip” when absolutely required. This way, you maintain a measure of control over replacement element definitions for as long as possible.

Once you move from “strict” to “lax” or from “lax” to “skip,” you are effectively decreasing the constraint granularity of the operation. This means that you cannot reverse this setting without risking impact upon consumer programs.

SUMMARY OF KEY POINTS

- In support of the Loose versioning strategy, XML Schema wildcards can be used to allow a schema to accept unknown content.
- When making changes to schemas that include wildcards, UPA violations are a constant factor.
- The setting of the xsd:any element’s namespace and processContents attributes help determine the range of allowable unknown content.