II MCA

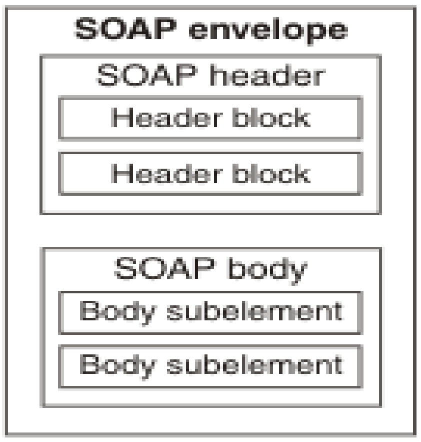
Web Technologies

Unit IV

SOAP and WSDL

The SOAP Model;

⇨ Web services are an instance of the **service-oriented architecture** pattern that use SOAP as the transport mechanism for moving messages between services described by WSDL interfaces



The architecture of a SOAP message

⇨ A SOAP message is encoded as an XML document, consisting of an **<Envelope>** element, which contains an optional **<Header>** element, and a mandatory **<Body>** element. The **<Fault>** element, contained in the <Body>, is used for reporting errors.

❖ **The SOAP envelope**

⇨ The SOAP <Envelope> is the root element in every SOAP message.

⇨ It contains two child elements, an optional <Header>, and a mandatory <Body>.

❖ **The SOAP header**

⇨ The SOAP <Header> is an optional subelement of the SOAP envelope.

⇨ It is used to pass application-related information that is to be processed by SOAP nodes along the message path.

❖ **The SOAP body**

⇨ The SOAP <Body> is a mandatory subelement of the SOAP envelope.

⇨ It contains information intended for the ultimate recipient of the message.

❖ **The SOAP fault**

⇨ The SOAP <Fault> is a subelement of the SOAP body, which is used for reporting errors.

⇨ With the exception of the <Fault> element, which is contained in the <Body> of a SOAP message,

⇨ XML elements in the <Header> and the <Body> are defined by the applications that make use of them.

An example of a SOAP 1.2 message

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
          env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
          env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
           env:mustUnderstand="true">
   <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itinerary
    xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>New York</p:departing>
     <p:arriving>Los Angeles</p:arriving>
     <p:departureDate>2001-12-14</p:departureDate>
     <p:departureTime>late afternoon</p:departureTime>
     <p:seatPreference>aisle</p:seatPreference>
   </p:departure>
   <p:return>
     <p:departing>Los Angeles</p:departing>
     <p:arriving>New York</p:arriving>
     <p:departureDate>2001-12-20</p:departureDate>
     <p:departureTime>mid-morning</p:departureTime>
     <p:seatPreference/>
   </p:return>
  </p:itinerary>
  <q:lodging
  xmlns:q="http://travelcompany.example.org/reservation/hotels">
  <q:preference>none</q:preference>
 </q:lodging>
 </env:Body>
</env:Envelope>
```

SOAP Message

➪ A SOAP message is an ordinary XML document containing the following elements −

- **Envelope** − Defines the start and the end of the message. It is a mandatory element.
- **Header** − Contains any optional attributes of the message used in processing the message, either at an intermediary point or at the ultimate end-point. It is an optional element.
- **Body** − Contains the XML data comprising the message being sent. It is a mandatory element.
- **Fault** − An optional Fault element that provides information about errors that occur while processing the message.

➪ **The following block depicts the general structure of a SOAP message −**

```
<?xml version = "1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV = "http://www.w3.org/2001/12/soap-envelope"
  SOAP-ENV:encodingStyle = "http://www.w3.org/2001/12/soap-encoding">
  <SOAP-ENV:Header>
    ...
    ...
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    ...
    ...
    <SOAP-ENV:Fault>
      ...
      ...
    </SOAP-ENV:Fault>
    ...
```

```
   </SOAP-ENV:Body>

</SOAP_ENV:Envelope>
```

SOAP Envelop

- ⇨ The SOAP Envelop is the container structure for the SOAP message and is associated with the namespace http://www.w3.org/2002/06/soap-envelop
- ⇨ The SOAP <Envelope> is the root element in every SOAP message.
- ⇨ It contains two child elements, an optional **<Header> element,** and a mandatory **<Body> element.**

```
<?xml version = "1.0"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV = " http://www.w3.org/2001/12/soap-envelope">

   <SOAP-ENV:Header>
      -----------------
-------------------------
   </SOAP-ENV:Header>

   ...   <SOAP-ENV:Body>
      -----------------
-------------------------
   </SOAP-ENV:BODY>

   ...
</SOAP-ENV:Envelope>
```

SOAP Header

- ⇨ The optional Header element offers a flexible framework for specifying additional application-level requirements.
- ⇨ For example, the Header element can be used to specify a digital signature for password-protected services.
- ⇨ Likewise, it can be used to specify an account number for pay-per-use SOAP services.
- ⇨ **SOAP Header Attributes**
  1. A SOAP Header can have the following two attributes −\
     - **Actor attribute**
     - **MustUnderstand attribute**
- ⇨ Actor attribute
  1. The SOAP protocol defines a message path as a list of SOAP service nodes. Each of these intermediate nodes can perform some processing and then forward the message to the next node in the chain. By setting the Actor attribute, the client can specify the recipient of the SOAP header.

- ⇨ **MustUnderstand attribute**

    1. It indicates whether a Header element is optional or mandatory. If set to true, the recipient must understand and process the Header attribute according to its defined semantics, or return a fault.

The following example shows how to use a Header in a SOAP message.

```
<?xml version = "1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV = " http://www.w3.org/2001/12/soap-envelope"
  SOAP-ENV:encodingStyle = " http://www.w3.org/2001/12/soap-encoding">

  <SOAP-ENV:Header>
    <t:Transaction
      xmlns:t = "http://www.tutorialspoint.com/transaction/"
          SOAP-ENV:role=" http://www.w3.org/2001/12/soap-envolpe/role/ultimateReceiver"
      SOAP-ENV:mustUnderstand = "true">5
    </t:Transaction>
  </SOAP-ENV:Header>
  ...
  ...
</SOAP-ENV:Envelope>
```

- ⇨ **The encodingStyle Attribute**

    1. Is used to declare how the contents of a header block were created
    2. Knowing this information allows a recipient of the header to decoder the information at it contains

SOAP Body

The required SOAP Body element contains the actual SOAP message intended for the ultimate endpoint of the message.

Immediate child elements of the SOAP Body element may be namespace-qualified.

Example

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body>
  <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
    <m:Item>Apples</m:Item>
  </m:GetPrice>
</soap:Body>
</soap:Envelope>
```

The example above requests the price of apples. Note that the m:GetPrice and the Item elements above are application-specific elements. They are not a part of the SOAP namespace.

A SOAP response could look something like this:

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body>
  <m:GetPriceResponse xmlns:m="http://www.w3schools.com/prices">
    <m:Price>1.90</m:Price>
  </m:GetPriceResponse>
</soap:Body>
</soap:Envelope>
```

SOAP Fault

The optional SOAP Fault element is used to indicate error messages.

If a Fault element is present, it must appear as a child element of the Body element. A Fault element can only appear once in a SOAP message.

The SOAP Fault element has the following sub elements:

Sub Element	**Description**
<faultcode>	A code for identifying the fault
<faultstring>	A human readable explanation of the fault
<faultactor>	Information about who caused the fault to happen
<detail>	Holds application specific error information related to the Body element

SOAP Fault Codes

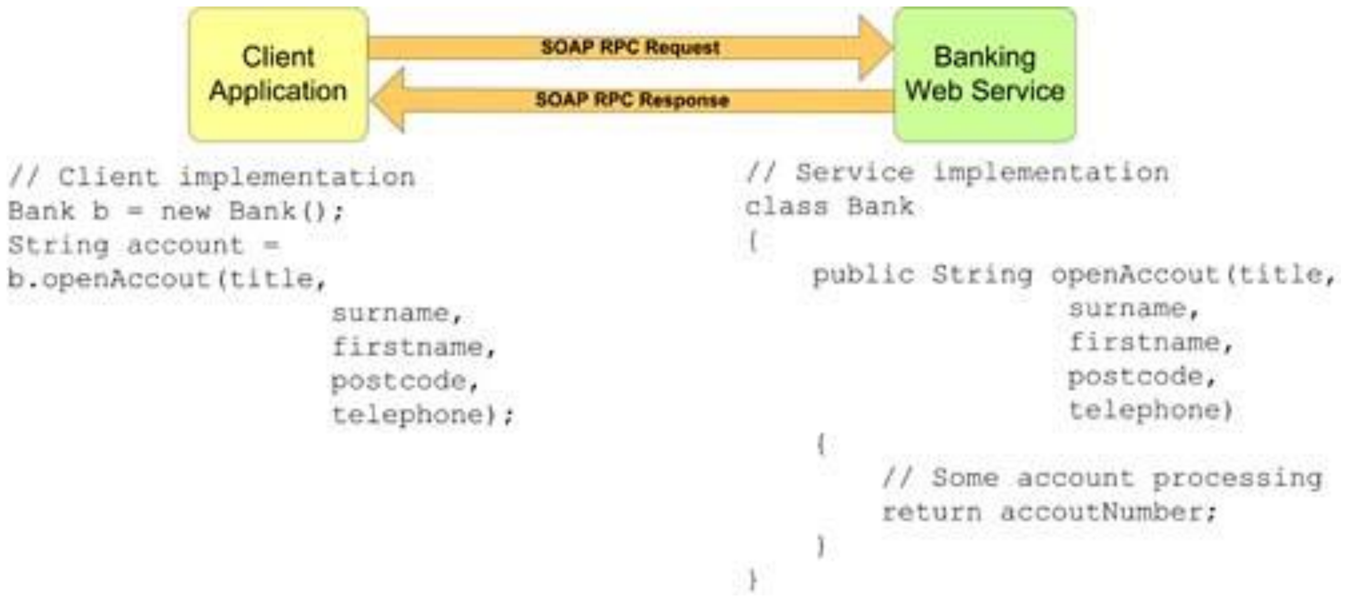
The faultcode values defined below must be used in the faultcode element when describing faults:

Error	**Description**
VersionMismatch	Found an invalid namespace for the SOAP Envelope element
MustUnderstand	An immediate child element of the Header element, with the mustUnderstand attribute set to "1", was not understood
Client	The message was incorrectly formed or contained incorrect information

Server	There was a problem with the server so the message could not proceed

SOAP RPC

- SOAP RPC provides toolkits with a convention for packaging SOAP-encoded messages so they can be easily mapped onto procedure calls in programming languages. To illustrate, let's return to our banking scenario and see how SOAP RPC might be used to expose account management facilities to users. Bear in mind throughout this simple example that it is an utterly insecure instance whose purpose is to demonstrate SOAP RPC only.
- Figure 3-16 shows a simple interaction between a Web service that offers the facility to open bank accounts and a client that consumes this functionality on behalf of a user. The Web service supports an operation called openAccount(…) which it exposes through a SOAP server and advertises as being accessible via SOAP RPC (SOAP does not itself provide a means of describing interfaces, but as we shall see later in the chapter, WSDL does). The client interacts with this service through a stub or proxy class called Bank which is toolkit-generated (though masochists are free to generate their own stubs) and deals with the marshalling and un-marshalling of local variables into SOAP RPC messages.



```
// Client implementation
Bank b = new Bank();
String account =
b.openAccout(title,
                      surname,
                      firstname,
                      postcode,
                      telephone);
```

```
// Service implementation
class Bank
{
   public String openAccout(title,
                            surname,
                            firstname,
                            postcode,
                            telephone)
   {
       // Some account processing
       return accoutNumber;
   }
}
```

Figure 3-17. A SOAP RPC request.

<?xml version="1.0" encoding="UTF-8"?> <env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"> <env:Body> <bank:openAccount env:encodingStyle= "http://www.w3.org/2002/06/soap-encoding" xmlns:bank="http://bank.example.org/account" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"> <bank:title xsi:type="xs:string"> Mr </bank:title> <bank:surname xsi:type="xs:string"> Bond </bank:surname> <bank:firstname xsi:type="xs:string"> James </bank:firstname> <bank:postcode xsi:type="xs:string"> S1 3AZ </bank:postcode> <bank:telephone xsi:type="xs:string"> 09876 123456 </bank:telephone> </bank:openAccount> </env:Body> </env:Envelope>

Figure 3-18. A SOAP RPC response.

<?xml version="1.0" encoding="UTF-8"?> <env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope"> <env:Body> <bank:openAccountResponse env:encodingStyle=

```
"http://www.w3.org/2002/06/soap-encoding" xmlns:rpc=         "http://www.w3.org/2002/06/soap-rpc"
xmlns:bank=         "http://bank.example.org/account" xmlns:xs=         "http://www.w3.org/2001/XMLSchema"
xmlns:xsi=         "http://www.w3.org/2001/XMLSchema-instance">
<rpc:result>bank:accountNo</rpc:result>          <bank:accountNo xsi:type="xsd:int">          10000014
</bank:accountNo>        </bank:openAccountResponse>     </env:Body> </env:Envelope>
```

Figure 3-19. SOAP RPC faults.

Fault	**SOAP Encoding for Fault**
Transient fault at receiver (e.g. out of memory error).	Fault with value of env:Receiver should be generated.
Receiver does not understand data encoding (e.g. encoding mechanism substantially different at sender and receiver.	A fault with a Value of env:DataEncodingUnknown for Code should be generated.
The service being invoked does not expose a method matching the name of the RPC element.	A fault with a Value of env:Sender for Code and a Value of rpc:ProcedureNotPresent for Subcode may be generated.
The reciever cannot parse the arguments sent. There may be too many or too few arguments, or there may be type mismatches.	A fault with a Value of env:Sender for Code and a Value of rpc:BadArguments for Subcode must be generated.

Figure 3-20. A SOAP RPC fault.

```
<?xml version="1.0"?> <env:Envelope          xmlns:env="http://www.w3.org/2002/06/soap-envelope"
xmlns:rpc="http://www.w3.org/2002/06/soap-rpc">     <env:Body>     <env:Fault>     <env:Code>
<env:Value>env:Sender</env:Value>                                                <env:Subcode>
<env:Value>rpc:BadArguments</env:Value>               </env:Subcode>             </env:Code>
<env:Reason>          Missing surname parameter          </env:Reason>     </env:Fault>     </env:Body>
</env:Envelope>
```

<u>Document, RPC, Literal, Encoded</u>

- **Document style:** The SOAP Body contains one or more child elements called parts. There are no SOAP formatting rules for what the body contains; it contains whatever the sender and the receiver agrees upon.
- **RPC style:** RPC implies that SOAP body contains an element with the name of the method or operation being invoked. This element in turn contains an element for each parameter of that method/operation.
- **SOAP Encoding:** SOAP encoding is a set of serialization. The rules specify how objects, structures, arrays, and object graphs should be serialized. Generally speaking, an application using SOAP encoding is focused on remote procedure calls and will likely use RPC message style. When SOAP encoding is used, the SOAP message contains data type information within the SOAP message. This makes serialization (data translation) easier since the data type of each parameter is denoted with the parameter.
- **Literal:** Data is serialized according to a schema. In practice, this schema is usually expressed using W3C XML Schema. The SOAP message does not directly contain any data type information, just a reference (namespace) to the schema that is used. To perform proper serialization (data translation) both, the sender and the receiver, must know the schema and must use the same rules for translating data.

The following SOAP message uses RPC style and SOAP encoding:

```
<soap:envelope>
```

```
   <soap:body>
      <myMethod>
         <x xsi:type="xsd:int">5</x>
         <y xsi:type="xsd:float">5.0</y>
      </myMethod>
   </soap:body>
</soap:envelope>
```

The following SOAP message uses RPC style and literal:

```
<soap:envelope>
   <soap:body>
      <myMethod>
         <x>5</x>
         <y>5.0</y>
      </myMethod>
   </soap:body>
</soap:envelope>
```

The following SOAP message uses document style and literal:

```
<soap:envelope>
   <soap:body>
      <x>5</x>
      <y>5.0</y>
   </soap:body>
</soap:envelope>
```

The following SOAP message uses document style and literal wrapped:

```
<soap:envelope>
   <soap:body>
      <myMethod>
         <x>5</x>
         <y>5.0</y>
      </myMethod>
   </soap:body>
</soap:envelope>
```

SOAP, Web Services, and the REST Architecture

- ⇨ **SOAP (Simple Object Access Protocol)** is a standards-based web services access protocol that has been around for a long time. Originally developed by Microsoft, SOAP isn't as simple as the acronym would suggest.s
- ⇨ **REST (Representational State Transfer)** is another standard, made in response to SOAP's shortcomings. It seeks to fix the problems with SOAP and provide a simpler method of accessing web services.

Below are the main differences between SOAP and REST

SOAP	REST

- SOAP stands for Simple Object Access Protocol

- REST stands for Representational State Transfer

- SOAP is a protocol. SOAP was designed with a specification. It includes a WSDL file which has the required information on what the web service does in addition to the location of the web service.

- REST is an Architectural style in which a web service can only be treated as a RESTful service if it follows the constraints of being
    1. Client Server
    2. Stateless
    3. Cacheable
    4. Layered System
    5. Uniform Interface

- SOAP cannot make use of REST since SOAP is a protocol and REST is an architectural pattern.

- REST can make use of SOAP as the underlying protocol for web services, because in the end it is just an architectural pattern.

- SOAP uses service interfaces to expose its functionality to client applications. In SOAP, the WSDL file provides the client with the necessary information which can be used to understand what services the web service can offer.

- REST use Uniform Service locators to access to the components on the hardware device. For example, if there is an object which represents the data of an employee hosted on a URL as http://demo.guru99 , the below are some of URI that can exist to access them

  http://demo.guru99.com/Employee

  http://demo.guru99.com/Employee/1

- SOAP requires more bandwidth for its usage. Since SOAP Messages contain a lot of information inside of it, the amount of data transfer using SOAP is generally a lot.

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV
="http://www.w3.org/2001/12/soap-envelope"
SOAP-ENV:encodingStyle
=" http://www.w3.org/2001/12/soap-encoding">
<soap:Body>
 <Demo.guru99WebService
 xmlns="http://tempuri.org/">
   <EmployeeID>int</EmployeeID>
   </Demo.guru99WebService>
 </soap:Body>
</SOAP-ENV:Envelope>
```

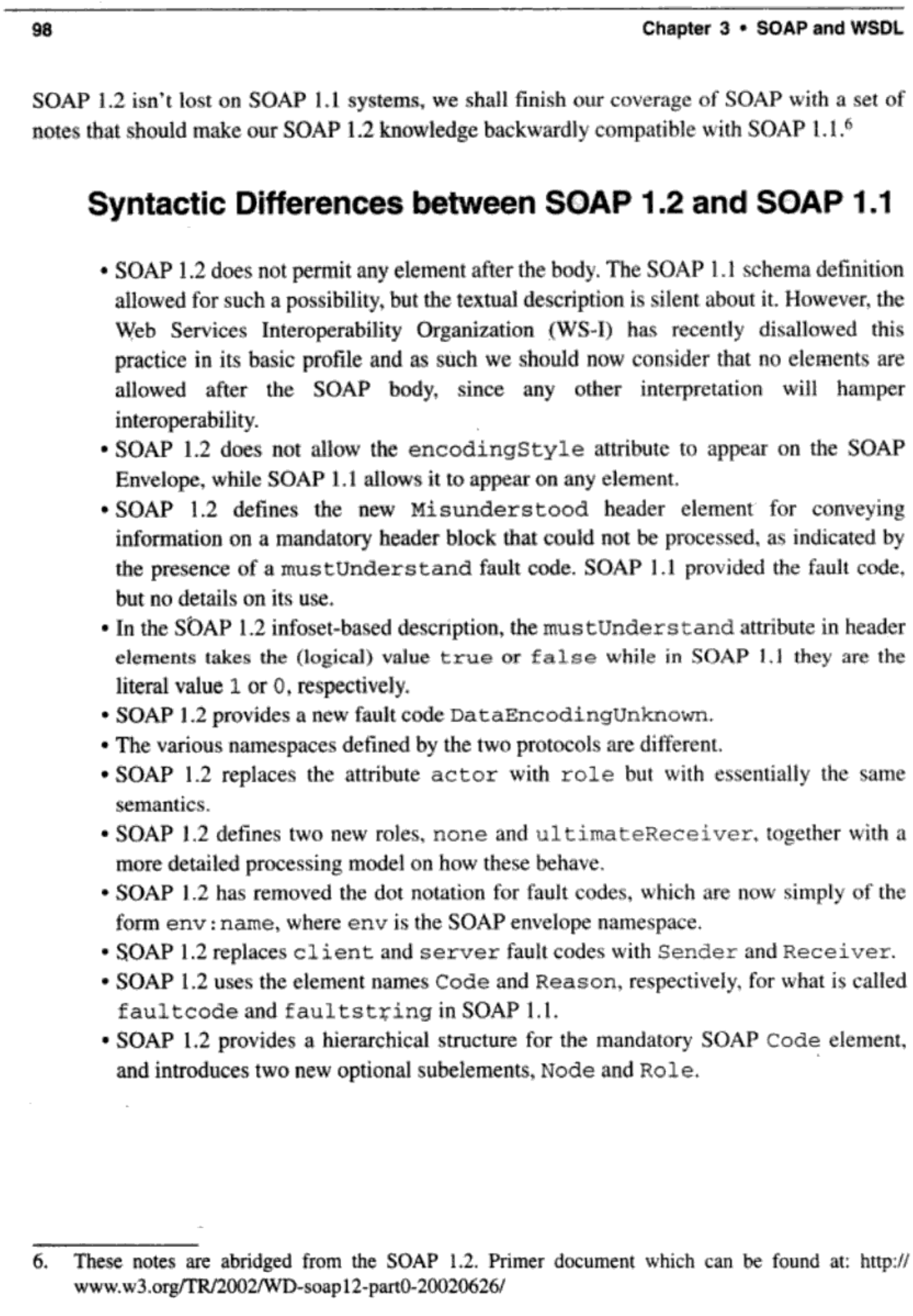
- REST does not need much bandwidth when requests are sent to the server. REST messages mostly just consist of JSON messages. Below is an example of a JSON message passed to a web server. You can see that the size of the message is comparatively smaller to SOAP.

  {"city":"Mumbai","state":"Maharastra"}

- SOAP can only work with XML format. As seen from SOAP messages, all data passed is in XML format.

- REST permits different data format such as Plain text, HTML, XML, JSON, etc. But the most preferred format for transferring data is JSON.

Syntactic Differences between SOAP 1.2 and SOAP 1.1



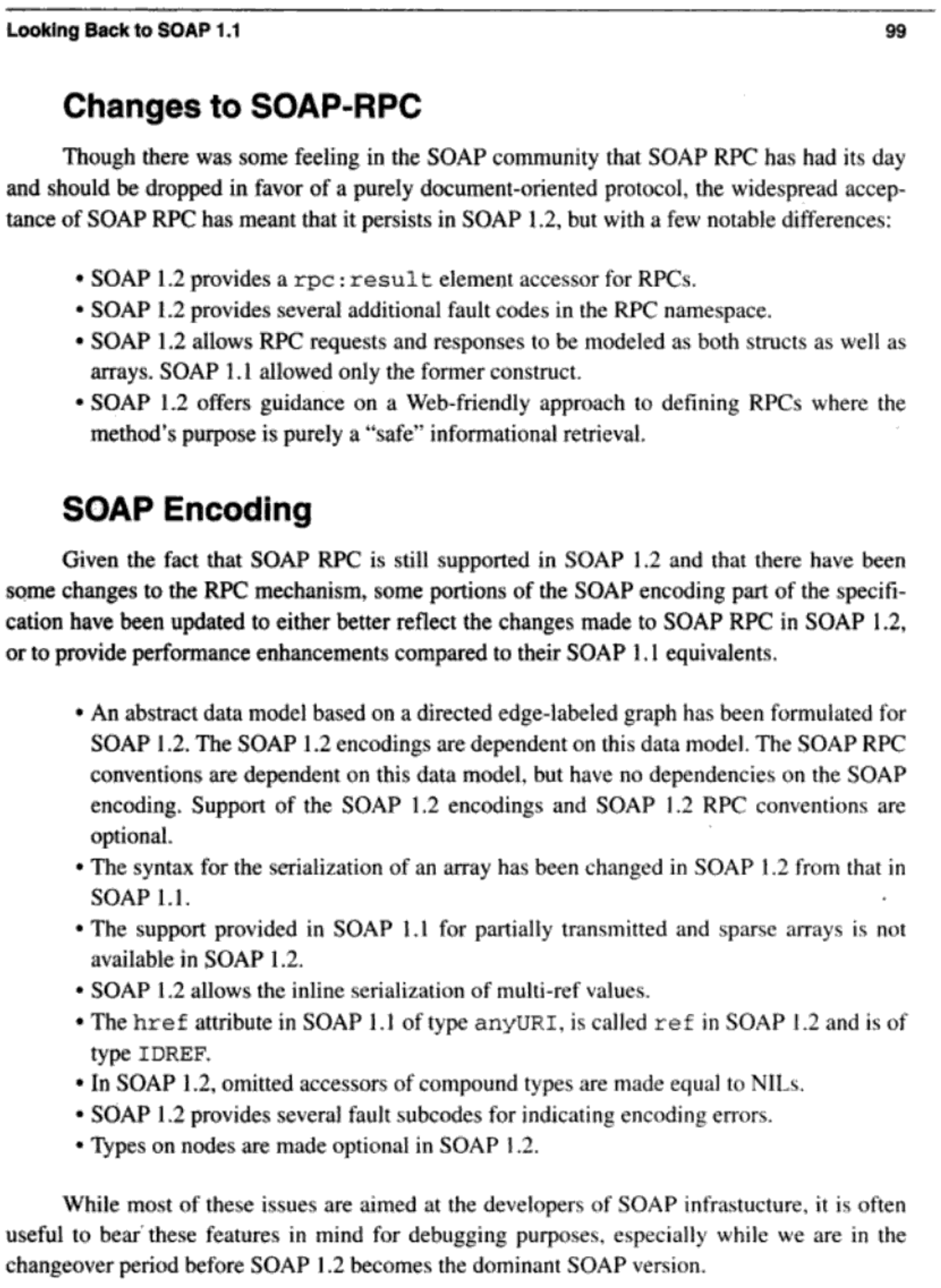
SOAP 1.2 isn't lost on SOAP 1.1 systems, we shall finish our coverage of SOAP with a set of notes that should make our SOAP 1.2 knowledge backwardly compatible with SOAP 1.1.[6]

Syntactic Differences between SOAP 1.2 and SOAP 1.1

- SOAP 1.2 does not permit any element after the body. The SOAP 1.1 schema definition allowed for such a possibility, but the textual description is silent about it. However, the Web Services Interoperability Organization (WS-I) has recently disallowed this practice in its basic profile and as such we should now consider that no elements are allowed after the SOAP body, since any other interpretation will hamper interoperability.
- SOAP 1.2 does not allow the `encodingStyle` attribute to appear on the SOAP Envelope, while SOAP 1.1 allows it to appear on any element.
- SOAP 1.2 defines the new `Misunderstood` header element for conveying information on a mandatory header block that could not be processed, as indicated by the presence of a `mustUnderstand` fault code. SOAP 1.1 provided the fault code, but no details on its use.
- In the SOAP 1.2 infoset-based description, the `mustUnderstand` attribute in header elements takes the (logical) value `true` or `false` while in SOAP 1.1 they are the literal value `1` or `0`, respectively.
- SOAP 1.2 provides a new fault code `DataEncodingUnknown`.
- The various namespaces defined by the two protocols are different.
- SOAP 1.2 replaces the attribute `actor` with `role` but with essentially the same semantics.
- SOAP 1.2 defines two new roles, `none` and `ultimateReceiver`, together with a more detailed processing model on how these behave.
- SOAP 1.2 has removed the dot notation for fault codes, which are now simply of the form `env:name`, where `env` is the SOAP envelope namespace.
- SOAP 1.2 replaces `client` and `server` fault codes with `Sender` and `Receiver`.
- SOAP 1.2 uses the element names `Code` and `Reason`, respectively, for what is called `faultcode` and `faultstring` in SOAP 1.1.
- SOAP 1.2 provides a hierarchical structure for the mandatory SOAP `Code` element, and introduces two new optional subelements, `Node` and `Role`.

6. These notes are abridged from the SOAP 1.2. Primer document which can be found at: http://www.w3.org/TR/2002/WD-soap12-part0-20020626/

Changes to SOAP RPC – SOAP Encoding



Changes to SOAP-RPC

Though there was some feeling in the SOAP community that SOAP RPC has had its day and should be dropped in favor of a purely document-oriented protocol, the widespread acceptance of SOAP RPC has meant that it persists in SOAP 1.2, but with a few notable differences:

- SOAP 1.2 provides a `rpc:result` element accessor for RPCs.
- SOAP 1.2 provides several additional fault codes in the RPC namespace.
- SOAP 1.2 allows RPC requests and responses to be modeled as both structs as well as arrays. SOAP 1.1 allowed only the former construct.
- SOAP 1.2 offers guidance on a Web-friendly approach to defining RPCs where the method's purpose is purely a "safe" informational retrieval.

SOAP Encoding

Given the fact that SOAP RPC is still supported in SOAP 1.2 and that there have been some changes to the RPC mechanism, some portions of the SOAP encoding part of the specification have been updated to either better reflect the changes made to SOAP RPC in SOAP 1.2, or to provide performance enhancements compared to their SOAP 1.1 equivalents.

- An abstract data model based on a directed edge-labeled graph has been formulated for SOAP 1.2. The SOAP 1.2 encodings are dependent on this data model. The SOAP RPC conventions are dependent on this data model, but have no dependencies on the SOAP encoding. Support of the SOAP 1.2 encodings and SOAP 1.2 RPC conventions are optional.
- The syntax for the serialization of an array has been changed in SOAP 1.2 from that in SOAP 1.1.
- The support provided in SOAP 1.1 for partially transmitted and sparse arrays is not available in SOAP 1.2.
- SOAP 1.2 allows the inline serialization of multi-ref values.
- The `href` attribute in SOAP 1.1 of type `anyURI`, is called `ref` in SOAP 1.2 and is of type `IDREF`.
- In SOAP 1.2, omitted accessors of compound types are made equal to NILs.
- SOAP 1.2 provides several fault subcodes for indicating encoding errors.
- Types on nodes are made optional in SOAP 1.2.

While most of these issues are aimed at the developers of SOAP infrastucture, it is often useful to bear these features in mind for debugging purposes, especially while we are in the changeover period before SOAP 1.2 becomes the dominant SOAP version.

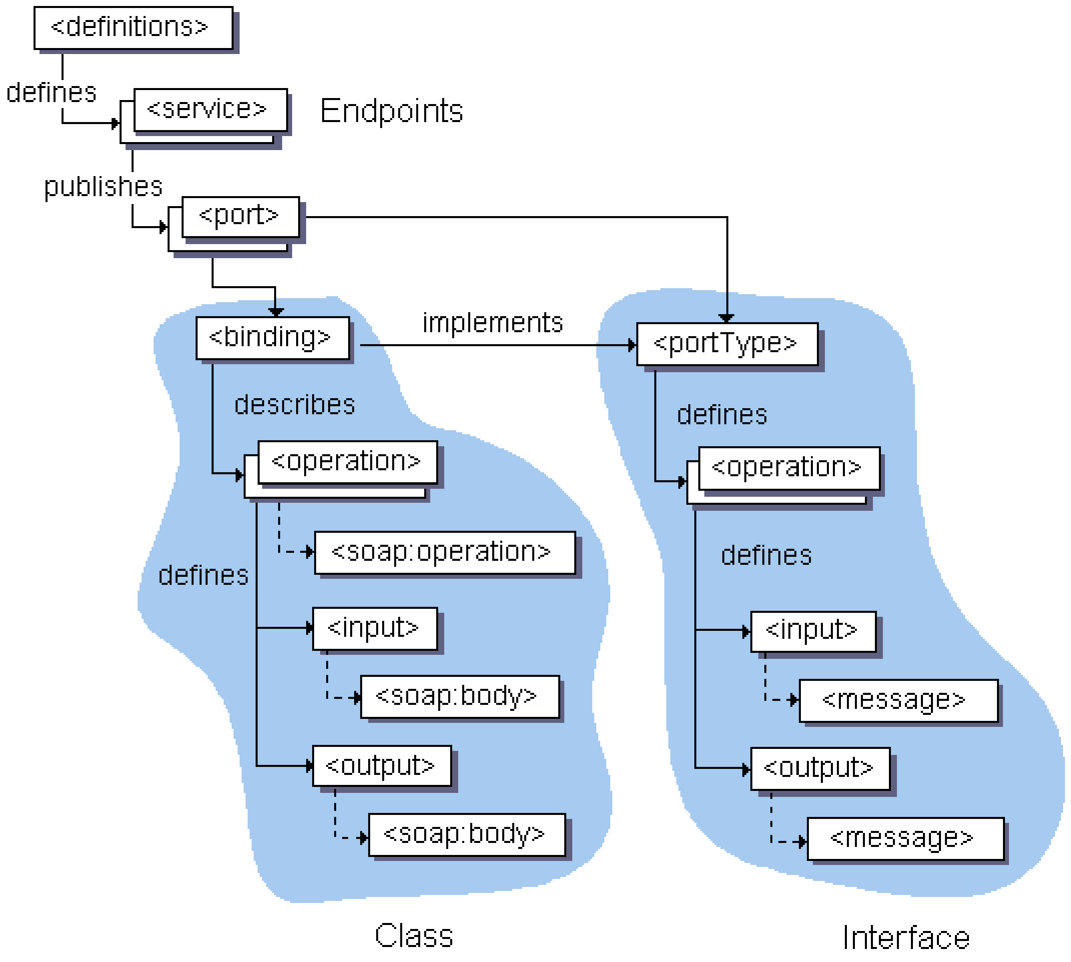
WSDL

- WSDL stands for Web Services Description Language
- WSDL is used to describe web services
- WSDL is written in XML
- WSDL is a W3C recommendation from 26. June 2007

WSDL Structure

⇨ Web Services Description Language (WSDL) is an XML grammar for describing network services as collections of communication endpoints capable of exchanging messages.

⇨ The diagram below illustrates the elements that are present in a WSDL document, and indicates their relationships



❖ **WSDL Document Elements**

⇨ A WSDL document has a definitions element that contains the other five elements, types, message, portType, binding and service. The following sections describe the features of the generated client code.

⇨ WSDL supports the XML Schemas specification (XSD) as its type system.

❖ **definitions**

Contains the definition of one or more services. JDeveloper generates the following attribute declarations for this section:

- name is optional.
- targetNamespace is the logical namespace for information about this service. WSDL documents can import other WSDL documents, and setting targetNamespace to a unique value ensures that the namespaces do not clash.
- xmlns is the default namespace of the WSDL document, and it is set to http://schemas.xmlsoap.org/wsdl/.
- All the WSDL elements, such as <definitions>, <types> and <message> reside in this namespace.
- xmlns:xsd and xmlns:soap are standard namespace definitions that are used for specifying SOAP-specific information as well as data types.
- xmlns:tns stands for this namespace.
- xmlns:ns1 is set to the value of the schema targetNamespace, in the <types> section.

Notice that the default of http://tempuri.org in namespaces to ensure that the namespaces are unique.

❖ **types**
Provides information about any complex data types used in the WSDL document. When simple types are used the document does not need to have a types section.

❖ **message**
An abstract definition of the data being communicated. In the example, the message contains just one part, response, which is of type string, where string is defined by the XML Schema.

❖ **operation**
An abstract description of the action supported by the service.

❖ **portType**
An abstract set of operations supported by one or more endpoints.

❖ **binding**
Describes how the operation is invoked by specifying concrete protocol and data format specifications for the operations and messages.

❖ **port**
Specifies a single endpoint as an address for the binding, thus defining a single communication endpoint.

❖ **service**
Specifies the port address(es) of the binding. The service is a collection of network endpoints or ports.

❖ WSDL Documents

An WSDL document describes a web service. It specifies the location of the service, and the methods of the service, using these major elements:

Element	**Description**
<types>	Defines the (XML Schema) data types used by the web service
<message>	Defines the data elements for each operation
<portType>	Describes the operations that can be performed and the messages involved.
<binding>	Defines the protocol and data format for each port type

The main structure of a WSDL document looks like this:

```
<definitions>

<types>
  data type definitions........
```

```
</types>

<message>
  definition of the data being communicated....
</message>

<portType>
  set of operations......
</portType>

<binding>
  protocol and data format specification....
</binding>

</definitions>
```

- **WSDL Example**

This is a simplified fraction of a WSDL document:

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

The stock quote WSDL interface,

- **Defintions**

⇨ It is the root element of all WSDL documents. It defines the name of the web service, declares multiple namespaces used throughout the remainder of the document, and contains all the service elements described here.

```
<definitions name="HelloService"
  targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
 ...............................................
</definitions>
```

❖ **The Types Elements**

⇨ A web service needs to define its inputs and outputs and how they are mapped into and out of the services.

⇨ WSDL **<types>** element takes care of defining the data types that are used by the web service. Types are XML documents, or document parts.

- The *types* element describes all the data types used between the client and the server.
- WSDL is not tied exclusively to a specific typing system.
- WSDL uses the W3C XML Schema specification as its default choice to define data types.
- If the service uses only XML Schema built-in simple types, such as strings and integers, then *types* element is not required.
- WSDL allows the types to be defined in separate elements so that the types are reusable with multiple web services.

a *types* element can be used within a WSDL.

```
<types>
  <schema targetNamespace = "http://example.com/stockquote.xsd"
    xmlns = "http://www.w3.org/2000/10/XMLSchema">

    <element name = "TradePriceRequest">
      <complexType>
        <all>
          <element name = "tickerSymbol" type = "string"/>
        </all>
      </complexType>
    </element>

    <element name = "TradePrice">
      <complexType>
        <all>
          <element name = "price" type = "float"/>
        </all>
      </complexType>
    </element>

  </schema>
</types>
```

❖ **Message Elements**

⇨ The **<message>** element describes the data being exchanged between the web service providers and the consumers.

- Each Web Service has two messages: input and output.
- The input describes the parameters for the web service and the output describes the return data from the web service.

- Each message contains zero or more **<part>** parameters, one for each parameter of the web service function.
- Each **<part>** parameter associates with a concrete type defined in the **<types>** container element.

Let us take a piece of code from the WSDL Example

```
<message name = "SayHelloRequest">
  <part name = "firstName" type = "xsd:string"/>
</message>

<message name = "SayHelloResponse">
  <part name = "greeting" type = "xsd:string"/>
</message>
```

❖ **PortType Elements**

⇨ The <portType> element defines **a web service**, the **operations** that can be performed, and the **messages** that are involved.

The request-response type is the most common operation type, but WSDL defines four types:

Type	**Definition**
One-way	The operation can receive a message but will not return a response
Request-response	The operation can receive a request and will return a response
Solicit-response	The operation can send a request and will wait for a response
Notification	The operation can send a message but will not wait for a response

WSDL One-Way Operation

A one-way operation example:

```
<message name="newTermValues">
  <part name="term" type="xs:string"/>
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="setTerm">
    <input name="newTerm" message="newTermValues"/>
  </operation>
</portType >
```

In the example above, the portType "glossaryTerms" defines a one-way operation called "setTerm".

The "setTerm" operation allows input of new glossary terms messages using a "newTermValues" message with the input parameters "term" and "value". However, no output is defined for the operation.

WSDL Request-Response Operation

A request-response operation example:

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

Bindings

⇨ The **<binding>** element provides specific details on how a *portType* operation will actually be transmitted over the wire.

- The bindings can be made available via multiple transports including HTTP GET, HTTP POST, or SOAP.
- The bindings provide concrete information on what protocol is being used to transfer *portType* operations.
- The bindings provide information where the service is located.
- For SOAP protocol, the binding is **<soap:binding>**, and the transport is SOAP messages on top of HTTP protocol.
- You can specify multiple bindings for a single *portType*.

⇨ The binding element has two attributes : *name* and *type* attribute.

```
<binding name = "Hello_Binding" type = "tns:Hello_PortType">
```

⇨ The *name* attribute defines the name of the binding, and the type attribute points to the port for the binding, in this case the "tns:Hello_PortType" port.

SOAP Binding

⇨ WSDL 1.1 includes built-in extensions for SOAP 1.1. It allows you to specify SOAP specific details including SOAP headers,

⇨ SOAP encoding styles, and the SOAPAction HTTP header. The SOAP extension elements include the following −

- soap:binding

- soap:operation
- soap:body

❖ **soap:binding**

⇨ This element indicates that the binding will be made available via SOAP. The *style* attribute indicates the overall style of the SOAP message format. A style value of *rpc* specifies an RPC format.
⇨ The *transport* attribute indicates the transport of the SOAP messages.
⇨ The value http://schemas.xmlsoap.org/soap/http indicates the SOAP HTTP transport, whereas http://schemas.xmlsoap.org/soap/smtp indicates the SOAP SMTP transport.

❖ **soap:operation**

⇨ This element indicates the binding of a specific operation to a specific SOAP implementation. The *soapAction* attribute specifies that the SOAPAction HTTP header be used for identifying the service.

❖ **soap:body**

⇨ This element enables you to specify the details of the input and output messages. In the case of HelloWorld, the body element specifies the SOAP encoding style and the namespace URN associated with the specified service.

Here is the piece of code

```
<binding name = "Hello_Binding" type = "tns:Hello_PortType">
  <soap:binding style = "rpc" transport = "http://schemas.xmlsoap.org/soap/http"/>
  <operation name = "sayHello">
    <soap:operation soapAction = "sayHello"/>

    <input>
      <soap:body
        encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
        namespace = "urn:examples:helloservice" use = "encoded"/>
    </input>

    <output>
      <soap:body
        encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
        namespace = "urn:examples:helloservice" use = "encoded"/>
    </output>
  </operation>
</binding>
```

Services

⇨ The **<service>** element defines the ports supported by the web service. For each of the supported protocols, there is one port element. The service element is a collection of ports.

- Web service clients can learn the following from the service element −
    - where to access the service,
    - through which port to access the web service, and
    - how the communication messages are defined.

- The service element includes a documentation element to provide human-readable documentation.

Here is a piece of code

```
<service name = "Hello_Service">
  <documentation>WSDL File for HelloService</documentation>
  <port binding = "tns:Hello_Binding" name = "Hello_Port">
    <soap:address
      location = "http://www.examples.com/SayHello/">
  </port>
</service>
```

- ⇨ The binding attributes of *port* element associate the address of the service with a binding element defined in the web service. In this example, this is *Hello_Binding*

```
<binding name =" Hello_Binding" type = "tns:Hello_PortType">
  <soap:binding style = "rpc"
    transport = "http://schemas.xmlsoap.org/soap/http"/>
  <operation name = "sayHello">
    <soap:operation soapAction = "sayHello"/>

    <input>
      <soap:body
        encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
        namespace = "urn:examples:helloservice" use = "encoded"/>
    </input>

    <output>
      <soap:body
        encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
        namespace = "urn:examples:helloservice" use = "encoded"/>
    </output>
  </operation>
</binding>
```

```
<wsdl:service name="StockBrokerService">
  <wsdl:port name="StockBrokerServiceSOAPPort"
   binding="tns:StockBrokerServiceSOAPBinding">
    <soap:address
        location="http://stock.example.org/"/>
  </wsdl:port>
</wsdl:service>
```

Figure 3-34 A `service` element declaration.

Managing WSDL Descriptions

While the service element is the final piece in the puzzle as far as an individual WSDL document goes, that's not quite the end of the story. For simple one-off Web services, we may choose to have a single WSDL document that combines both concrete and abstract parts of the interface. However, for more complex deployments we may choose to split the abstract parts into a separate file, and join that with a number of different concrete bindings and services to better suit the access pattern for those services.

For example, it may be the case that a single abstract definition (`message`, `portType`, and `operation` declarations) might need to be exposed to the network via a number of protocols, not just SOAP. It might also be the case that a single protocol endpoint might need to be replicated for quality of service reasons or perhaps even several different organizations each want to expose the same service as part of their Web service offerings. By using the WSDL import mechanism, the same abstract definition of the service functionality can be used across all of these Web services irrespective of the underlying protocol or addressing. This is shown in Figure 3-35 where MIME, HTTP, and SOAP endpoints all share the same abstract functionality yet expose that functionality to the network each in their own way. Additionally, the SOAP protocol binding has been deployed at multiple endpoints which can be within a single administrative domain or spread around the whole Internet and yet each service, by dint of the fact that they share the same abstract definitions, is equivalent.

If a WSDL description needs to include features from another WSDL description or an external XML Schema file, then the `import` mechanism is used. It behaves in a similar fashion to the XML Schema `include` feature where it can be used to include components from other WSDL descriptions. We have already seen how the WSDL import mechanism is used in Figure 3-27 where the XML Schema types from the stockbroker schema were exposed to the stock broking WSDL description, as follows:

```
<wsdl:import namespace="http://stock.example.org/schema"
  location="http://stock.example.org/schema"/>
```

The `import` feature of WSDL means that a WSDL description can leverage existing XML infrastructure—previously defined schemas for in-house documents, database schemas, existing Web services, and the like—without having to reproduce those definitions as part of its own description.

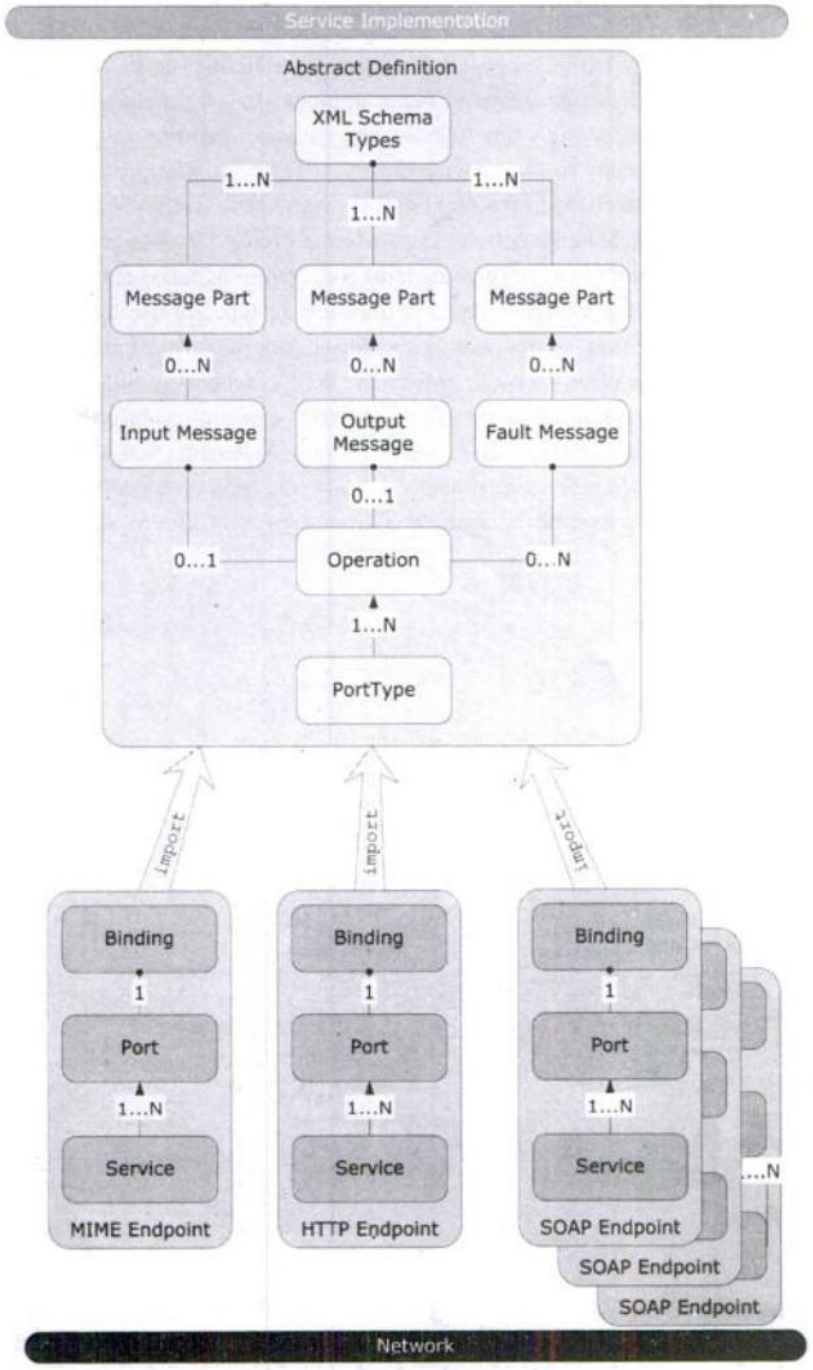


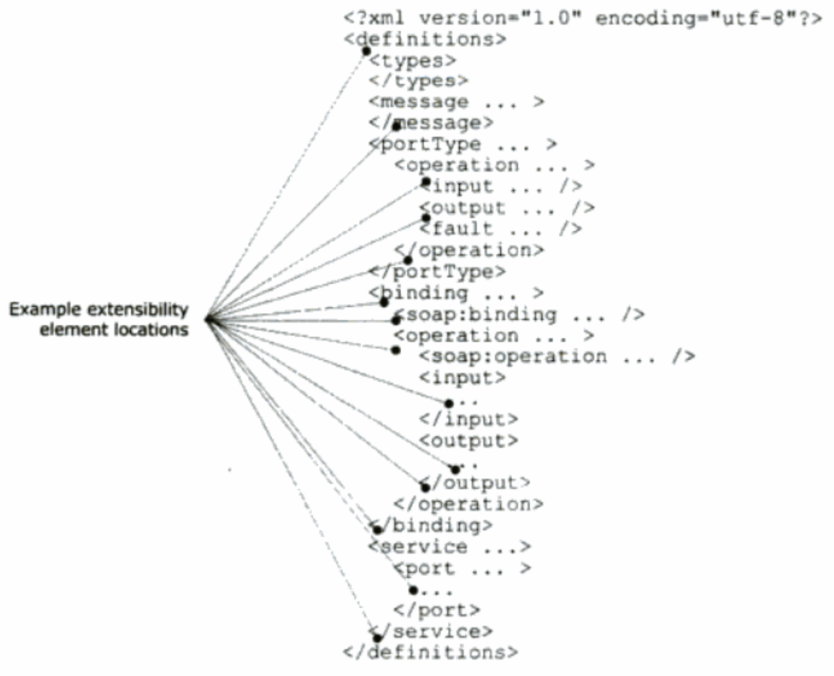
Figure 3-35 Including abstract WSDL descriptions for concrete endpoints.

Extending WSDL

Extending WSDL[8]

As Web services technology has advanced and matured, WSDL has begun to form the basis of higher-level protocols that leverage the basic building blocks that it provides, to avoid duplication of effort. Many of the technologies that we are going to examine throughout this book extend WSDL via such means to their own purpose. However, where SOAP offers header blocks as its extensibility mechanism for higher-level protocols to use, WSDL offers extension elements based on the XML Schema notion of substitution groups (see Chapter 2).

In the WSDL schema, several (abstract) global element declarations serve as the heads of substitution groups. In addition, the WSDL schema defines a base type for use by extensibility elements as a helper to ensure that the necessary substitution groups are present in any extensions. While it is outside the scope of this book to present the WSDL schema in full, there exists in the schema extensibility elements which user-defined elements can use to place themselves at *any* point within a WSDL `definition`. There are extensibility elements that allow extensions to appear at global scope, within a `service` declaration, before the `port` declaration, in a `message` element before any `part` declarations and any other point in a WSDL description, as shown in Figure 3-36.



```
<?xml version="1.0" encoding="utf-8"?>
<definitions>
  <types>
  </types>
  <message ... >
  </message>
  <portType ... >
    <operation ... >
      <input ... />
      <output ... />
      <fault ... />
    </operation>
  </portType>
  <binding ... >
    <soap:binding ... />
    <operation ... >
      <soap:operation ... />
      <input>
        ...
      </input>
      <output>
        ...
      </output>
    </operation>
  </binding>
  <service ...>
    <port ... >
      ...
    </port>
  </service>
</definitions>
```

Figure 3-36 WSDL substitution group heads.

8. This section based on a draft version of the WSDL 1.2 specification.

For example, the `soap` elements that we have seen throughout the bindings section of our WDSL description are extensibility elements. In the schema for those elements, they have been declared as being part of the substitution group `bindingExt` which allows them to legally appear as part of the WSDL `bindings` section.

Additionally, third-party WSDL extensions may declare themselves as mandatory with the inclusion of a `wsdl:required` attribute in their definitions. Once a `required` attribute is set, any and all validation against an extended WSDL document must include the presence of the corresponding element as a part of the validation.

Extensibility elements are commonly used to specify some technology-specific binding. They allow innovation in the area of network and message protocols without having to revise the base WSDL specification. WSDL recommends that specifications defining such protocols also define any necessary WSDL extensions used to describe those protocols or formats.[9]

Using SOAP and WSDL

While many of the more advanced features of the emerging Web services architecture are still being built into many of the platforms, support for SOAP and WSDL in most vendors' Web services toolkits is widespread and makes binding to and using Web services straightforward. In this section, we investigate how a typical application server and can be used to deploy our simple banking example, and how it can be later consumed by a client application. The overall architecture can be seen in Figure 3-37.

The architecture for this sample is typical of Web services applications that routinely combine a variety of platforms. In Figure 3-37, we use Microsoft's .Net and Internet Information Server to host the service implementation, but we use the Java platform and the Apache AXIS Web service toolkit to consume this service and drive the application.

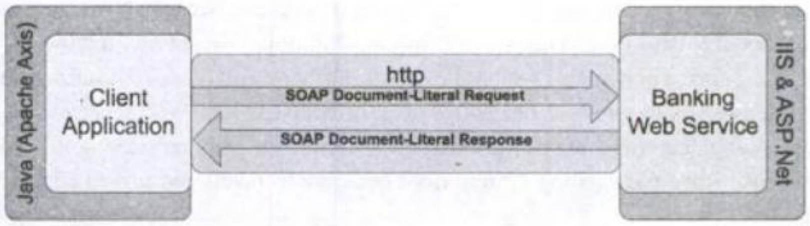


Figure 3-37 Cross-platform banking Web service example.

9. From WSDL 1.2 specification, http://www.w3.org/TR/wsdl12/.

Service Implementation and Deployment

The implementation of our banking service is a straightforward C# class, and is shown in Figure 3-38.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Web;
using System.Web.Services;

[WebService(Namespace="http://bank.example.org")]
public class BankService : System.Web.Services.WebService
{
  [WebMethod]
  public string openAccount(string title,
                            string surname,
                            string firstname,
                            string postcode,
                            string telephone)
  {
    BankEndSystem bes = new BackEndSystem();
    string accountNumber = bes.processApplication(title,
                                                  surname,
                                                  firstname,
                                                  postcode,
                                                  telephone);
    return accountNumber;
  }
}
```

Figure 3-38 A simple bank Web service implementation.

Most of the work for this service is done by some back-end banking system, to which our service delegates the workload. Our service implementation just acts as a kind of gateway between the Web service network to which it exposes our back-end business logic, and the back-end systems themselves to which it delegates work it receives from Web services clients. This pattern is commonplace when exposing existing systems via Web services, and makes good architectural sense since the existing system does not have to be altered just to add in Web service support.

Unit V

UDDI

UDDI at a Glance

- ⇨ UDDI is an acronym for Universal Description, Discovery and Integration.
- ⇨ The UDDI is a registry and a protocol for publishing and discovering Web services.
- ⇨ UDDI is the associated **standards-based, open, and platform-independent** means of publishing and locating these services.
- ⇨ UDDI is a XML based framework for describing, discovering and integrating web services.
- ⇨ UDDI is a directory of web service interfaces described by WSDL, containing information about web services.
- ⇨ UDDI communicate via SOAP
- ⇨ UDDI is built into **Microsoft.et platform**
- ⇨ UDDI uses WSDL to describe interfaces to web services
- ⇨ UDDI uses **World Web Consortium (W3C) and Internet Engineering Task Force (IETF)** Internet standard such as XML, HTTP and DNS protocol

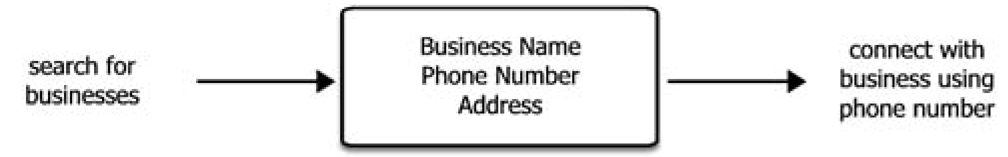
Analogies with Telephone Directories

- ⇨ UDDI is a directory of web services that are available from different vendors
- ⇨ UDDI provides a means of **adding new services, removing existing services and changing the contact (endpoint) information** for services
- ⇨ Both telephone directories and UDDI registries provide a means to locate a vendor or provider of a particular service.
- ⇨ For telephone directories, contact information is basically a phone number and perhaps may also include an address. Contact information in a UDDI registry consists of information about the service provider as well as technical information about the Web service itself.
- ⇨ Conceptually, the information available in an UDDI registry is similar to that in the **white, green, and yellow pages** of the phone book. In UDDI, the segmentation of information that is available and searchable can be thought of as follows:

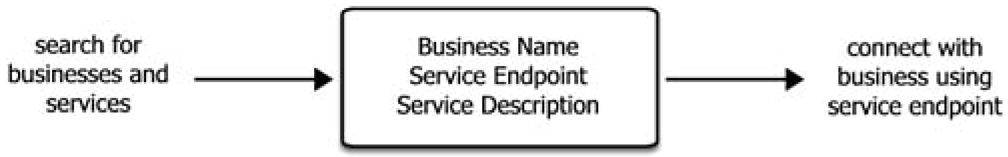
  - ➢ **White Pages**:
    - o Contact information about the service provider company. This information includes the business or entity name, address, contact information, other short descriptive information about the service provider, and unique identifiers with which to facilitate locating this business.

- **Yellow Pages:**
  - Categories (taxonomies) under which Web services implementing functionalities within those categories can be found.
- **Green Pages**:
  - Technical information about the capabilities and behavioral grouping of Web services.

Similarities between (a) telephone directory books and (b) UDDI registries

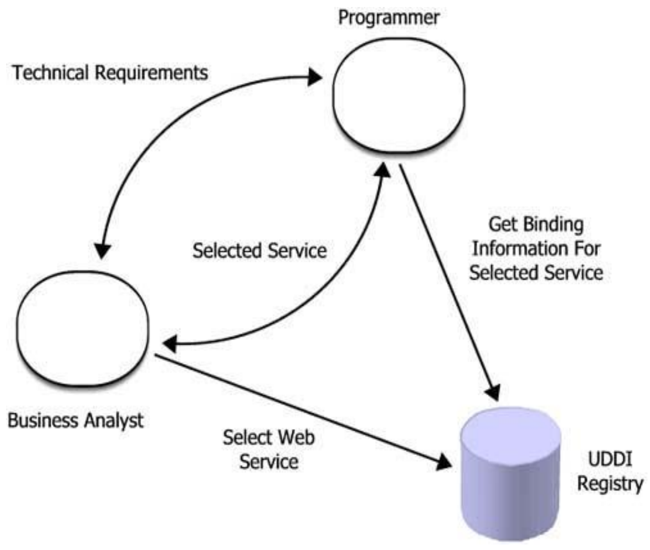


(a) Telephone Directory Book


search for businesses and services
Business Name
Service Endpoint
Service Description
connect with business using service endpoint

(b) UDDI Registry

The typical roles played while interacting with an UDDI registry



The UDDI Business Registry (UBR)

- ⇨ The UDDI Business Registry (UBR) is a global implementation of the UDDI specification.
- ⇨ The UBR is a single registry for Web services.
- ⇨ A group of companies operate and host UBR nodes, each of which is an identical copy of all other nodes
- ⇨ New entries or updates are entered into a single node, but are propagated to all other nodes.
- ⇨ The UBR is a key element of the deployment of Web services and provides the following capabilities:

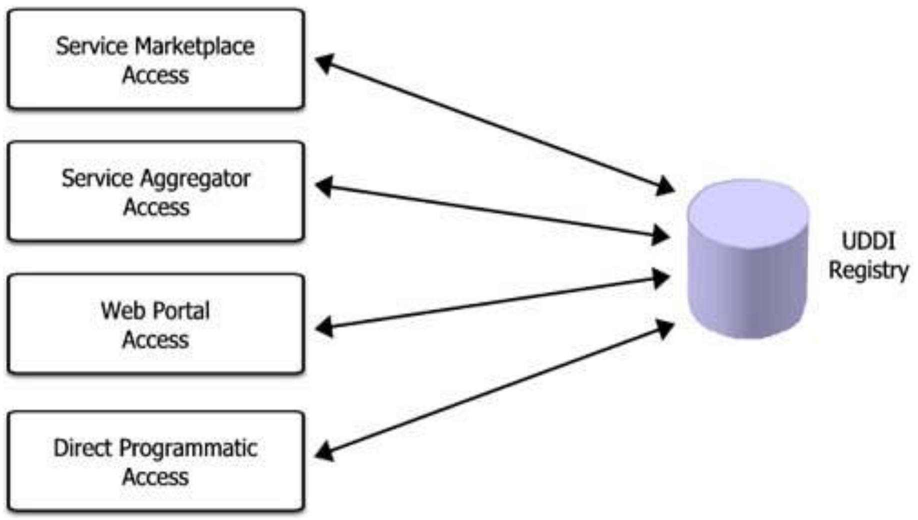
    - A centralized registration facility at which to publish and make others aware of the Web services a company makes available.
    - A centralized search facility at which companies that require a particular service can locate businesses that provide that service as well as relevant information about that service.

- ⇨ A small group of companies operate and manage a set of UBR nodes.

    1. In July 2002, the UBR was updated to support version 2 of the UDDI specification. Initially, IBM, Microsoft, and SAP comprised the UBR V2, operating 3 UBR nodes.
    2. NTT Communications later launched an UBR node to become the fourth UBR V2 node.
    3. More than 10,000 businesses are registered with the initial three UBR nodes, publishing over 7,000 Web services.
    4. NTT expects to add another 1,000 businesses within the first operational year of the fourth UBR node.

depicts some typical means of accessing and interacting with an UDDI registry.

The various means of accessing an UDDI registry.


Service Marketplace Access
Service Aggregator Access
Web Portal Access
Direct Programmatic Access
UDDI Registry

UDDI Under the Covers

❖ The UDDI Specification

⇨ Version 3 is the most recent incarnation of the UDDI specification.

⇨ Version 3 builds on and expands the foundations laid by versions 1 and 2 of the UDDI specification, and presents a blueprint for flexible and interoperable Web services registries. Version 3 also includes a rich set of enhancements as well as additional features, including improved security and new APIs.

The major documents of the UDDI Specification version 3.

UDDI Version 3 Specification Documents	Synopsis
Features List	Brief overview of the key features in version 3.
Specification	The actual specification document.
XML Schemas	A set of XML Schema files that formally describe UDDI data structures.
WSDL Service Interface Descriptions	A set of files that describe the UDDI Version 3 WSDL interface definitions.

❖ UDDI Core Data Structures

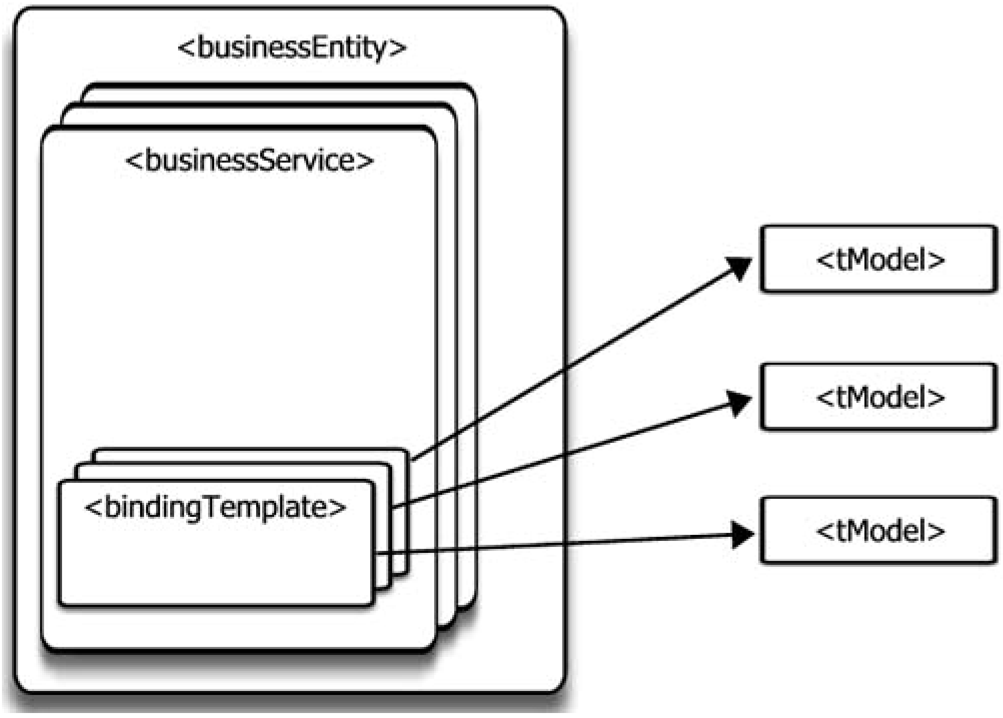
⇨ Information representation within UDDI consists of instances of persistent data structures that are expressed in XML.

⇨ It is these data structures that are persistently stored and managed by UDDI nodes.

The different entity types defined by the UDDI Information Model.

Entity Type Name	Description
businessEntity	A business that provides a Web service.
businessService	A collection of related services offered by a business.
bindingTemplate	Technical information about a particular Web service.
tModel	Technical model information about a Web service that is used to determine whether a service is compatible with the client's needs.

The interrelationship between the UDDI core data structures.



\

- ❖ **<businessEntity>**
- ⇨ The businessEntity entity type represents information about service providers within UDDI.
- ⇨ This information includes detailed data about the name of the provider, contact information, and some other short descriptions of the provider.
- ⇨ This information may also be provided in multiple languages.
- ⇨ One or more of the businessService entity types are contained within a businessEntity structure and represents information about the services offered by that businessEntity.

- ❖ **<businessService>**
- ⇨ The businessService entity type is a logical grouping of Web services and provides information about the bundled purpose of a set of contained Web services.
- ⇨ One or more of the bindingTemplate entity types are contained within a businessService structure and provides technical information about a particular Web service.

- ❖ **<bindingTemplate>**
- ⇨ The bindingTemplate structure directly or indirectly provides descriptive technical information about an instance of a Web service, and includes a network location or endpoint of the service.
- ⇨ The network location (access point) is usually a URL, but can be other network access points such as email addresses.
- ⇨ The bindingTemplate structure also includes information about the type of Web service located at that access point through references to tModel entities as well as other parameters.

- **<tModels>**

- tModels, which are short for technical models, provide more detailed information about a Web service.
- tModels are reusable entities that are referenced from bindingTemplate structures and denote compliance with a shared concept or design.
- The set of tModels that a bindingTemplate refers to makes up a Web service's technical fingerprint. The actual documents and information identified by a tModel are not located within the UDDI registry itself, but instead the tModel provides pointers to the location where such documents can be found.
- Two more UDDI entity types that are important are **subscription** and **publisherAssertion**.
- The ***subscription*** entity type describes the request to keep track of the evolution or changes to particular entities.
- The ***publisherAssertion*** entity type describes the relationship between one businessEntity and another businessEntity.
- There are many instances where multiple divisions within a large organization or a group of organizations want to make the relationship between them known in order to facilitate discovery of the services they provide.
- The individual divisions or organizations each have their own businessEntity, and the entity type publisherAssertion describes the relationship between two businessEntity structures.
- It is important to note that two organizations must assert the same relationship through the publisherAssertion for that relationship to be publicly available.
- This disallows the situation where one organization claims a relationship with another where in fact there is none.

Accessing UDDI

- UDDI is itself a Web service and as such, applications can communicate with an UDDI registry by sending and receiving XML messages.
- This makes the access both language and platform independent.
- Although it's possible, it is unlikely that programmers will deal with the low-level details of sending and receiving XML messages.
- Instead, client-side packages for different languages and platforms will emerge that facilitate programmatic access to UDDI.
- Two such packages are UDDI4J and Microsoft's UDDI SDK, which are client-side APIs for communicating with UDDI from Java and .Net programs, respectively.
- UDDI4J was originally developed by IBM and released in early 2001 as an open source initiative. Later, HP joined and contributed to the initiative, developing much of the version 2 release

Figure 4-6 shows a complete application using UDDI4J to connect to Microsoft's UDDI Business Registry (UBR) inquiry node and locate service providers whose name includes the string "abc". After an `UDDIProxy` proxy object for Microsoft's UBR inquiry node is set up, the `find_business` method is invoked to search for available business names that contain the substring "abc". The wildcard character '%' is used to specify that the substring may occur anywhere in the business name. Qualifiers, such as case-sensitive string matching, could have been added to the `find_business` method to further limit the search.

```
/**
 * The AccessUDDI class implements a simple application
 *  that connects to Microsoft's UBR inquiry node,
 *  searches for service providers that have the string
 *  "abc" in their name and displays to the standard
 *  output the business name, the business description,
 *  and the names of all services provided by that
 *  business.
 */

import org.uddi4j.client.UDDIProxy;
import org.uddi4j.datatype.Name;
import org.uddi4j.response.BusinessInfo;
import org.uddi4j.response.BusinessList;
import org.uddi4j.response.ServiceInfo;
import java.util.Vector;

public class AccessUDDI
{
   public static void main ( String[] args )
   {
      int i = 0;
      int j = 0;

      UDDIProxy proxy = new UDDIProxy ();
      try
      {
         // Set the inquiryURL
         proxy.setInquiryURL
            ( "http://uddi.microsoft.com/inquire" );

         // Look for names that include "abc"
         Vector names = new Vector ();
         names.add ( new Name ( "%abc%" ) );
```

Figure 4-6 Using UDDI4J to access an UDDI Registry to print out 21 providers that include the string "abc" in their names.

```
         // Search the UDDI registry
         BusinessList results =
            proxy.find_business (
               names,
               null,
               null,
               null,
               null,
               null,
               21 );

         Vector businessInfoVect = results.getBusinessInfos
().getBusinessInfoVector ();

         System.out.println ( "Results are:" );

         for ( i = 0 ; i < businessInfoVect.size () ; i++ )
         {
            BusinessInfo businessInfo = ( BusinessInfo )
businessInfoVect.elementAt ( i );

            System.out.println ( "\nName: " +
businessInfo.getNameString () );
            System.out.println ( " ... Description: " +
businessInfo.getDefaultDescriptionString () );

            Vector serviceInfoVect =
businessInfo.getServiceInfos ().getServiceInfoVector ();
            for ( j = 0 ; j < serviceInfoVect.size () ; j++ )
            {
               ServiceInfo servInfo = ( ServiceInfo )
serviceInfoVect.elementAt ( j );
               System.out.println ( " ... Service Name: " +
servInfo.getNameString () );
            }
         }
      }
      catch ( Exception e )
      {
         e.printStackTrace ();
      }
   }
}
```

Figure 4-6 Using UDDI4J to access an UDDI Registry to print out 21 providers that include the string "abc" in their names (continued).

Once the results of the search are returned from the UDDI registry, additional method calls are used to extract the business name, business description, and service names for all matching businesses. This information is then displayed on the standard output. Figure 4-7 shows a selected subset of the output of the application shown in Figure 4-6.

```
Results are:

Name: abc
 ... Description: null

Name: ABC Corporate Services
 ... Description: A travel services company serving the agent
and hotel segments of the industry.
 ... Service Name: Traveler's Emergency Service System (TESS)
 ... Service Name: Premier Hotel Program (PHP)
 ... Service Name: Global Connect

Name: abc Enterprise
 ... Description: test object
 ... Service Name: Deutsche Telekom Productshow
 ... Service Name: Deutsche Telekom Shopping
 ... Service Name: Deutsche Telekom T-Mobil
 ... Service Name: Deutsche Telekom T-Online

Name: abc inc
 ... Description: test desc

Name: ABC Insurance
 ... Description: null

Name: ABC Microsystems
 ... Description: ?~~ ?? ?? ??...???...
 ... Service Name: <New Service Name>

Name: ABC Music
 ... Description: null
 ... Service Name: List Instruments

Name: ABC travel agency
 ... Description: travel buses for goa,bombay,delhi.

Name: abc123
 ... Description: null
 ... Service Name: bogus service

Name: CompanyABC
 ... Description: null

Name: IntesaBci Sistemi e Servizi
 ... Description: IntesaBci Sistemi e Servizi co-ordinate all of
Bank IntesaBci's operations with regard to the development and
management of IT and telecommunication systems
 ... Service Name: Home Page
```

Figure 4-7 A subset of the result of running the application shown in Figure 4-6.

How UDDI is Playing Out

- ⇨ How UDDI will truly be used by companies will determine how, when, where, and why businesses will register their Web services.
- ⇨ UDDI has focused on its analogous behavior with standard telephone directory books: UDDI provides a listing of businesses and the services each business offers as well as a means of searching and discovering Web services to use within consuming applications.
- ⇨ Since this usage of UDDI is during the design of applications, it can be referred to as the design-time use.

❖ UDDI and Lifecycle Management

- ⇨ To understand the usefulness of UDDI at run time, consider the issues that developers and companies have to grapple with after they have developed a Web service or an application that consumes Web services.

The steps in this lifecycle management scenario proceed as follows:

1. Locate a Web service that fulfills the application's needs using whatever means that are useful, including portals, service aggregators, or programmatically with an UDDI registry directly.
2. If the Web service was not initially discovered within an UDDI registry, locate the same service within an UDDI registry and save (e.g., in a database) the bindingTemplate information.
3. Develop the application to consume the Web service using the information from the saved bindingTemplate information.
4. If the Web service call fails or exceeds an application-specified time-out, query the UDDI registry for the latest information on that Web service.
5. In case the original Web service call failed, compare the latest binding information for that Web service with the saved information. If the latest binding information for the Web service is different from the saved information, then save the new binding information, and retry the Web service call.
6. In the case that the original Web service call exceeded a time-out, compare the latest binding information for that Web service with the saved information. If the information is different or newer access endpoints are available, select another endpoint. The selection procedure may be manual in which the application allows the user to manually choose, or it may be automatic.

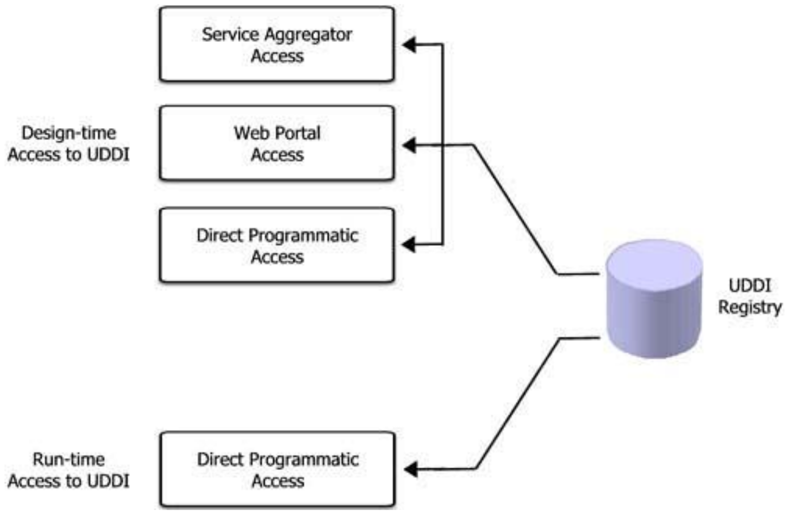
. Retrying Web service invocations based on dynamic UDDI information.

```
 // The Web service invocation failed, so check to see
//  whether new binding information is available. If so,
 //  retry the Web service call.
BindingDetail bd = proxy.get_bindingDetail ( bindingKey );
Vector btvect = bindingDetail.getBindingTemplateVector (); BindingTemplate bt = ( BindingTemplate )
btvect.elementAt ( 0 );
 newEndpoint = bt.getAccessPoint ().getText ();
if ( thisEndpoint.equalsIgnoreCase ( newEndpoint ) )
 {
 // In this case, the endpoint information has changed
//  so we should retry the Web service invocation with
 //  with the new endpoint
 thisEndpoint = newEndpoint;
 retry = true;
}
else {
 // In this case, the endpoint information has not
 //  changed so there no reason to retry the Web
 //  service invocation
 retry = false;
}
```

- **UDDI and Dynamic Access Point Management**

- Usually, this variety of service deployments is dynamic, that is, the Web service is initially deployed on a single server.
- Later, as the service becomes more popular and demand increases, additional access points are deployed.
- These deployments may be a cluster of servers in close proximity to each other, a geographically distributed set of servers, or both.\
- A client application that consumes the Web service may have been developed before the additional access points were deployed. Or the best service at the time the application was developed is no longer the best or the most appropriate.
- For example, the client application may have been developed in one country and later used in another country.

The use of UDDI at both design time and run time.


Service Aggregator Access
Design-time Access to UDDI
Web Portal Access
Direct Programmatic Access
UDDI Registry
Run-time Access to UDDI
Direct Programmatic Access

Conversations Overview

Web Services Conversation Language (WSCL)

- ⇨ Web Services Conversation Language (WSCL) allows the business level conversations or public processes supported by a Web service to be defined.
- ⇨ WSCL specifies the XML documents being exchanged, and the allowed sequencing of these document exchanges.
- ⇨ WSCL conversation definitions are themselves XML documents and can therefore be interpreted by Web Services infrastructures and development tools.
- ⇨ **The rationale behind WSCL is that it captures the "time" dimension of Web services conversations and allows Web services to declare the conversation pattern through which they can be driven. WSCL is, at the time of writing, a W3C note submitted by the Hewlett-Packard Company.**
- ⇨ **In January 2003, along with the process modeling language WSCI,[1] WSCL became one of the inputs for the W3C's Choreography working group.**

❖ Consuming WSCL Interfaces

- ⇨ Like WSDL, WSCL is an interface language designed to be consumed by Web services toolkits.
- ⇨ Using both WSDL and WSCL, it is possible for toolkits to create proxies for Web services that not only encapsulate the remote operations and SOAP serialization aspects, but also provide structured help on the order in which operations should be invoked.
- ⇨ That is, while WSDL provides message format and operation information, the WSCL description supports the creation of a conversation state machine that can guide the consumer of a Web service through its use.
- ⇨ For example, consider the Java interfaces shown in

A "Static" interface

```
 public boolean login(String user, String pass);
public boolean buySong(String title, String artist);
 public File pay(CreditCard cc);
```

A more "Conversational" interface.

```
 public boolean login(String user, String pass);
public boolean buySong(String title, String artist)
                              throws NotLoggedInException;
public File pay(CreditCard cc)
                             throws NoSongBoughtException;
```

A simple WSCL-based proxy implementation.

```
 public boolean login(String user, String pass)
{
// Login to service...   _
loggedIn = // The result of logging into the service
return _loggedIn;
}
 public boolean buySong(String title, String artist)
                    throws NotLoggedInException
{
 if(!_loggedIn)
{
        throw new NotLoggedInException();
 }
 // Buy song logic
 _token = // The token from the remote service   return true;
}
public File pay(CreditCard cc)
                throws NoSongBoughtException
{
 if(_token == null)
{
                throw new NoSongBoughtException();
}
// Retrieve the file
return file;
 }
```

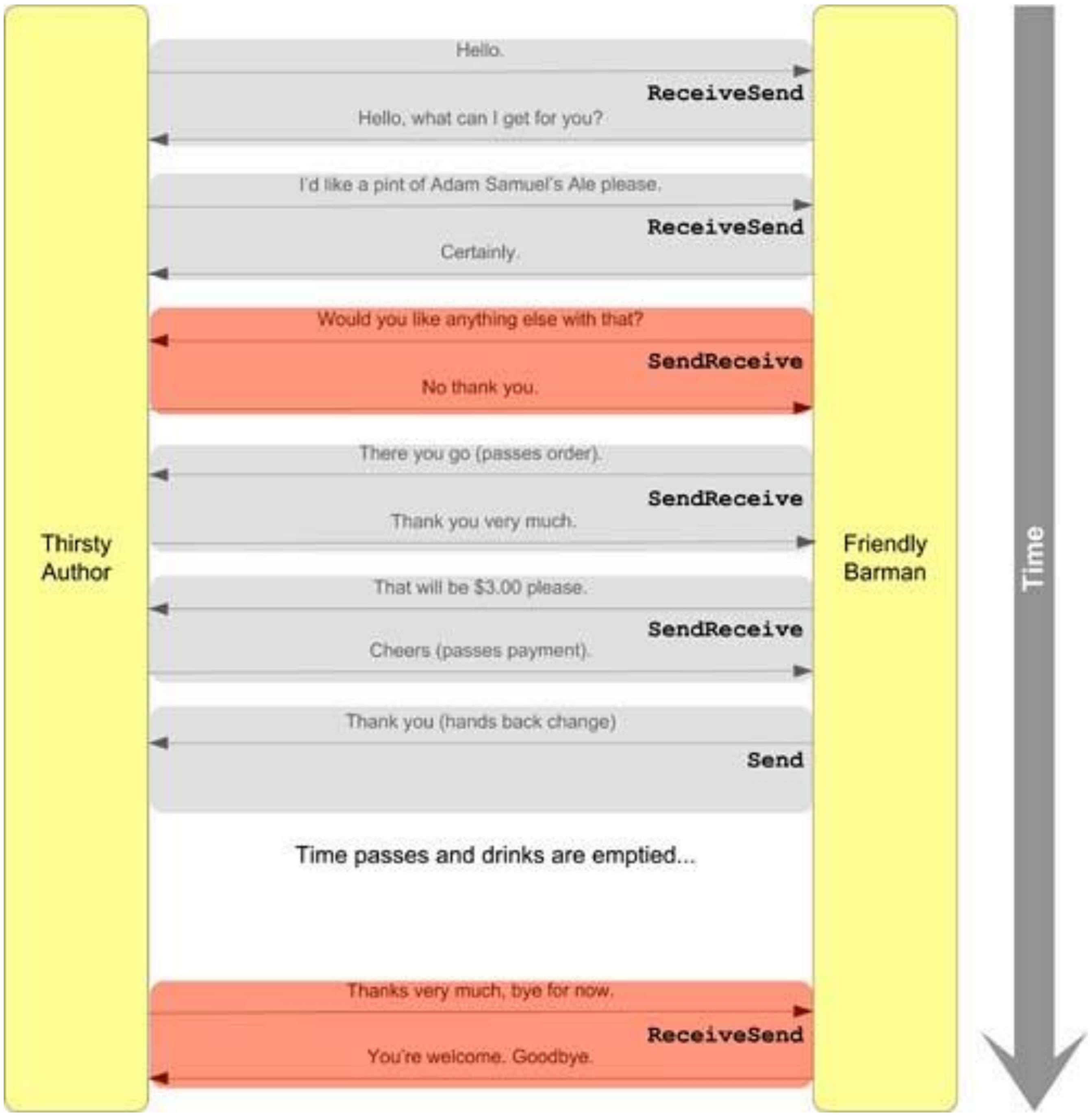
<u>WSCL Interface Components</u>

- ⇨ A WSCL interface is both complimentary and similar in spirit to its associated WSDL description.
- ⇨ Like WSDL, a WSCL interface starts off simple and gradually builds up its description stage by stage until a whole conversation pattern is formed, ready for consumption on the Web.
- ⇨ Our example, we will start at the abstract sections of the WSCL interface and work our way through to a full-fledged conversation.

❖ Interactions

- ⇨ An interaction with a conversational Web service is modeled as an XML document exchange whose flow is seen from the point of view of the service being invoked.
- ⇨ WSCL supports four distinct message exchange patterns.
  1. **Send :-** The service creates a one-way message that is sent to the consumer and expects no correlated response from that consumer
  2. **Receive :-** The service expects to receive a message from the consumer, while the consumer expects no message back in return
  3. **SendReceive :-** The service initiates a bilateral message exchange with the consumer, for which it expects a conrrelated response
  4. **ReceiveSend :-** The service expects to receive a message from the consumer to which it will respond

Determining the interaction types of the conversation.


Thirsty Author
Friendly Barman
Time
Hello.
ReceiveSend
Hello, what can I get for you?
I'd like a pint of Adam Samuel's Ale please.
ReceiveSend
Certainly.
Would you like anything else with that?
SendReceive
No thank you.
There you go (passes order).
SendReceive
Thank you very much.
That will be $3.00 please.
SendReceive
Cheers (passes payment).
Thank you (hands back change)
Send
Time passes and drinks are emptied...
Thanks very much, bye for now.
ReceiveSend
You're welcome. Goodbye.

Defining the "Beer" schema.

```
 <?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
xmlns:b="http://drinks.example.org/beer"
targetNamespace=" ">
<xs:complexType name="beer">
<xs:sequence>
<xs:element name="name" type="xs:string"/>
<xs:element name="brewer" type="xs:string"/>
</xs:sequence>
<xs:attribute name="domestic" type="xs:boolean"      use="required"/>
```

```
</xs:complexType>
<xs:element name="beer" type="b:beer"/>
 </xs:schema>
```

Defining the "Money" schema.

```
 <?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
xmlns:m="http://money.example.org"   targetNamespace="http://money.example.org">
<xs:complexType name="money">
<xs:sequence>
  <xs:element name="currency" type="xs:string"/>
<xs:element name="value" type="xs:decimal"/>
</xs:sequence>
</xs:complexType>
 <xs:element name="money" type="m:money"/>
</xs:schema>
```

The CustomerGreetingMessage schema.

```
 <?xml version="1.0" encoding="UTF-8"?>
<xs:schema
targetNamespace="http://conversations.example.org/bar/CustomerGreetingMessage"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
<xs:element name="CustomerGreetingMessage">
<xs:simpleType>      <xs:restriction base="xs:string">
<xs:enumeration value="Hello"/>
 </xs:restriction>
 </xs:simpleType>
</xs:element>
</xs:schema>
```

The StaffGreetingMessage schema.

```
 <?xml version="1.0" encoding="UTF-8"?>
<xs:schema                                targetNamespace="http://conversations.example.org/bar/StaffGreetingMessage"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
<xs:element name="StaffGreetingMessage">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="Hello, how may I help you?"/>
</xs:restriction>
 </xs:simpleType>
</xs:element>
</xs:schema>
```

The CustomerOrderMessage schema.

```
 <?xml version="1.0" encoding="UTF-8"?>
<xs:schema                              targetNamespace="http://conversations.example.org/bar/CustomerOrderMessage"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
xmlns:b="http://drinks.example.org/beer"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
<xs:element name="CustomerOrderMessage" type="b:beer"/>
 </xs:schema>
```

The BillMessage schema.

```
 <?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://conversations.example.org/bar/BillMessage"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:m="http://money.example.org" elementFormDefault="qualified"
attributeFormDefault="unqualified">
<xs:element name="BillMessage" type="m:money"/>
 </xs:schema>
```

The general form of a WSCL interaction.

```
 <Interaction interactionType="<send, receive, etc>"        >
 <InboundXMLDocument hrefSchema="<schema URI>"/>
<OutboundXMLDocument hrefSchema="<schema URI>"        />
</Interaction>
```

- ⇨ The interactions between the endpoints are specified in terms of the message we previously defined. Where in our human conversation we might expect the exchange along the lines of:
- ⇨ Thirsty author: Hello. Friendly barman: Hello, what can I get you?
- ⇨ We would now expect their (rather less personal) WSCL interaction equivalents:

```
 <Interaction interactionType="ReceiveSend" >
<InboundXMLDocument   hrefSchema="http://conversations.example.org/bar/CustomerGreetingMessage"      />
<OutboundXMLDocument  hrefSchema="http://conversations.example.org/bar/StaffGreetingMessage"          />
</Interaction>
```

- ⇨ Similarly, instead of the human interaction:
- ⇨ Friendly barman: That will be $3.00 please. Thirsty author: Cheers (passes money to barman)
- ⇨ we have the WSCL interaction that captures this action:

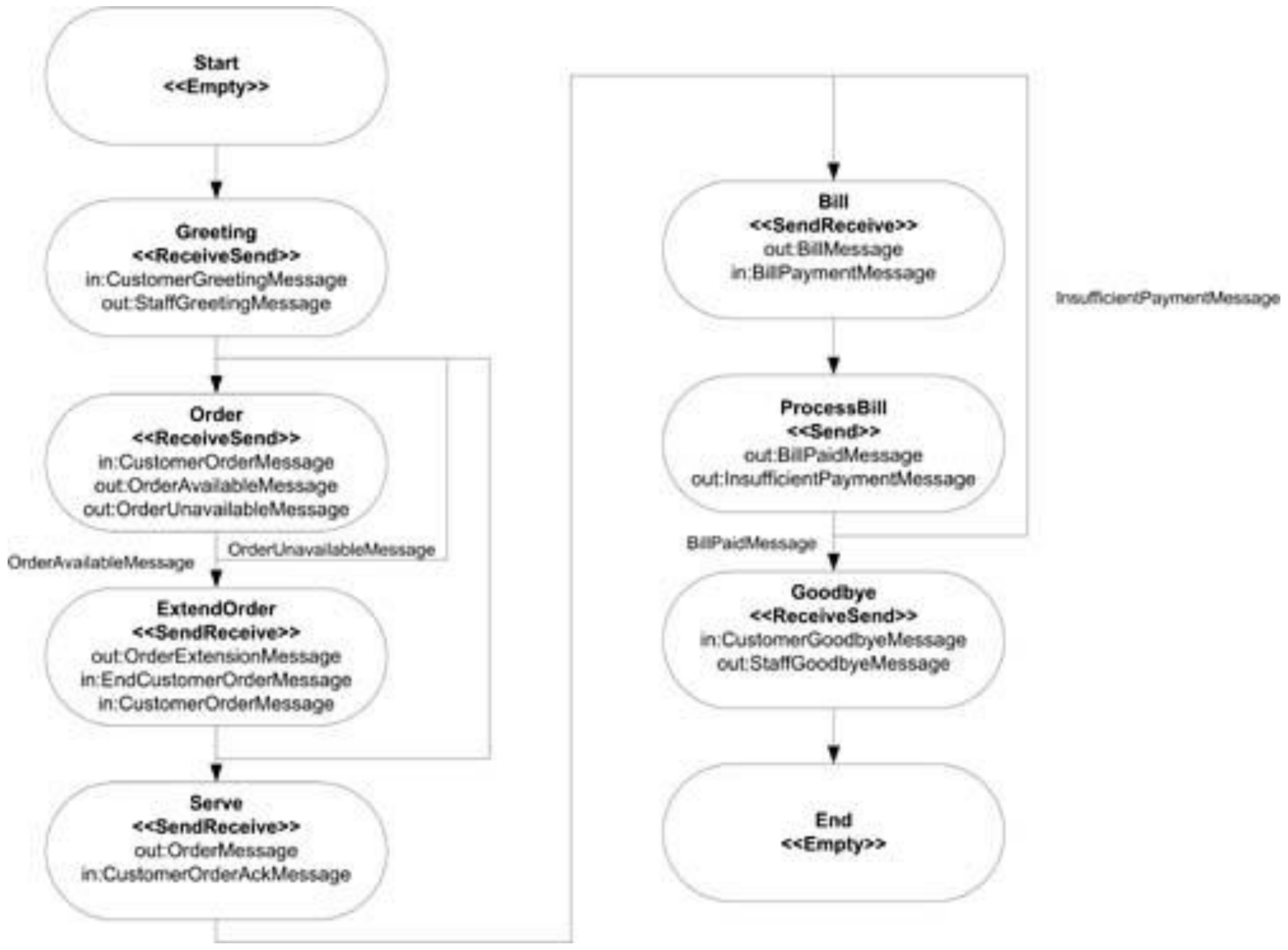
```
 <Interaction interactionType="SendReceive" >
<OutboundXMLDocument    hrefSchema="http://conversations.example.org/bar/BillMessage"                  />
<InboundXMLDocument  hrefSchema="http://conversations.example.org/bar/BillPaymentMessage"              />
</Interaction>
```

- ❖ **Transitions**

- ⇨ Having implemented our conversation types, its message set and its set of interactions, we can now go one stage further with our implementation and actually design the transitions that implement the state machine for this conversation that is, actually get down to the business of ordering the interactions into a full-fledged conversation.
- ⇨ For this, we use the communication pattern derived from Figure, which we have redrawn for clarity as a UML Activity Diagram

⇨ The activity diagram shows the possible states of the conversation and highlights the message exchanges at each state, along with the interaction patter.

A UML activity diagram for the bar conversation.



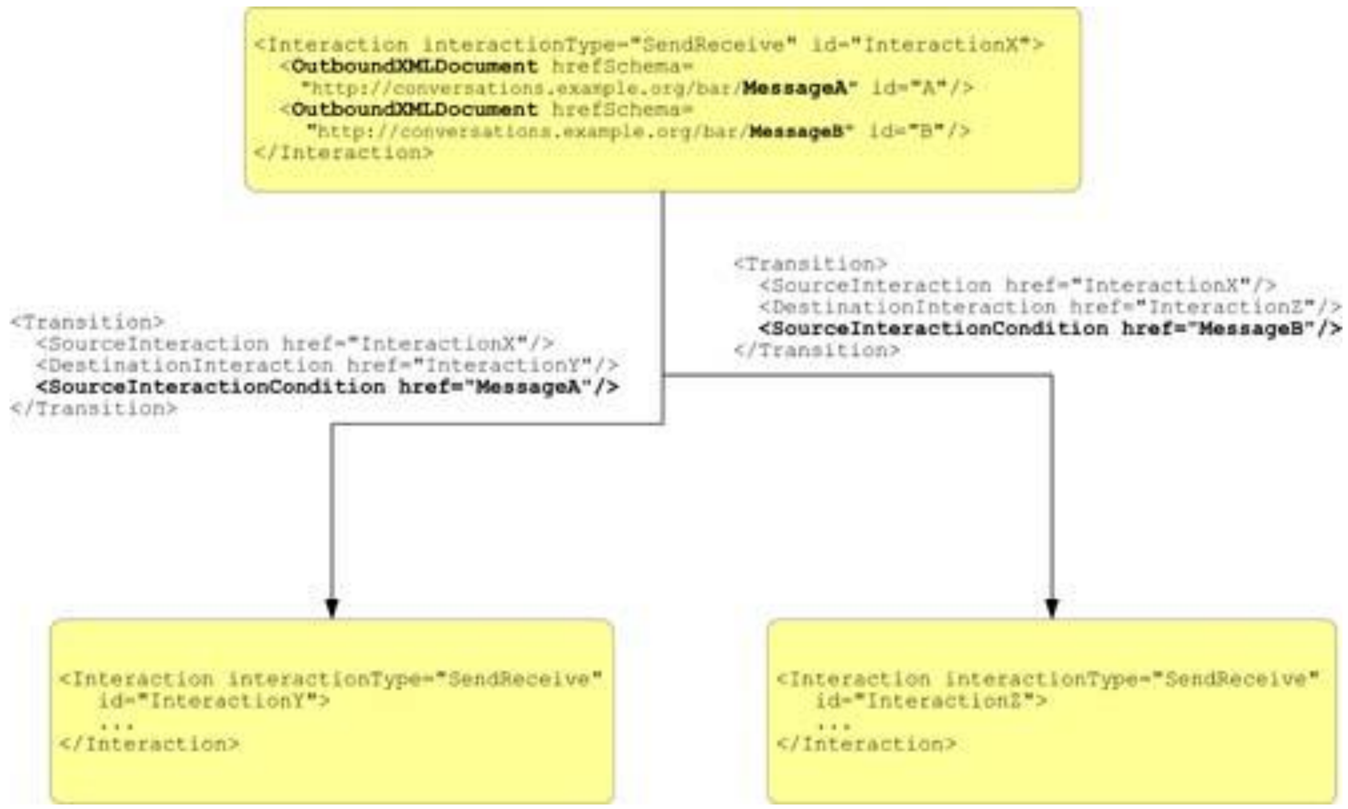
⇨ Codifying such rules in WSCL is straightforward. Each Transition element in a WSCL document has the following form:

```
 <Transition>
<SourceInteraction href="{some interaction}"/>
<DestinationInteraction href="{another interaction}"/>
<SourceInteractionCondition href="{some message}"/>     ... more conditions
</Transition>
```

⇨ Where the Transition element is a container for the following items:

- A previous interaction that has occurred referenced by a SourceInteraction element.
- The next interaction to occur references by the DestinationInteraction element.
- A number of SourceInteractionCondition elements that only allow this transition to occur if the referenced message was the last document sent (by the Web service) during the interaction.

Guarding state transitions with SourceInteractionCondition elements.



```
<Interaction interactionType="SendReceive" id="InteractionX">
  <OutboundXMLDocument hrefSchema=
    "http://conversations.example.org/bar/MessageA" id="A"/>
  <OutboundXMLDocument hrefSchema=
    "http://conversations.example.org/bar/MessageB" id="B"/>
</Interaction>
```

```
<Transition>
  <SourceInteraction href="InteractionX"/>
  <DestinationInteraction href="InteractionY"/>
  <SourceInteractionCondition href="MessageA"/>
</Transition>
```

```
<Transition>
  <SourceInteraction href="InteractionX"/>
  <DestinationInteraction href="InteractionZ"/>
  <SourceInteractionCondition href="MessageB"/>
</Transition>
```

```
<Interaction interactionType="SendReceive"
   id="InteractionY">
   ...
</Interaction>
```

```
<Interaction interactionType="SendReceive"
   id="InteractionZ">
   ...
</Interaction>
```

❖ Conversations

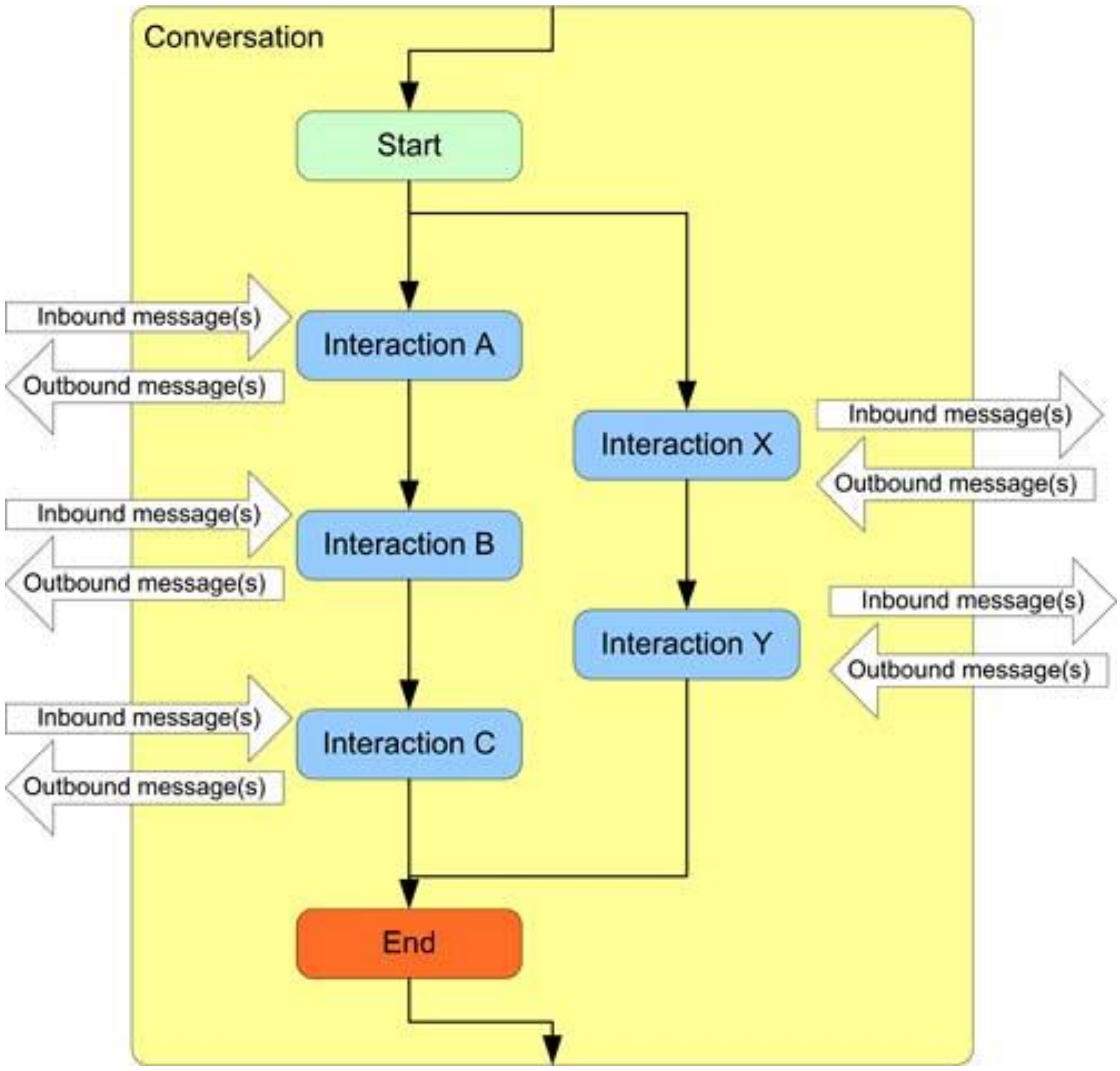
- ⇨ The Conversation part of the WSCL description draws together the messages, interactions, and transitions into the final conversation description.
- ⇨ Conversation part of WSCL begins with its endpoints, how conversations are begun and ended.
- ⇨ The first and last interactions of the conversation are given special identifiers in the WSCL specification such that a software agent consuming a WSCL interface can readily determine how to start a conversation and figure out when it has finished.
- ⇨ If we examine the conversation element that supports our bar example, we see the following (abbreviated) opening tag for the Conversation element:

```
<Conversation name="BarConversation"   initialInteraction="Start"   finalInteraction="End">


<Interaction interactionType="Empty" />
```

- ⇨ A possible conversation graph that shows this multiplexing behavior is presented in

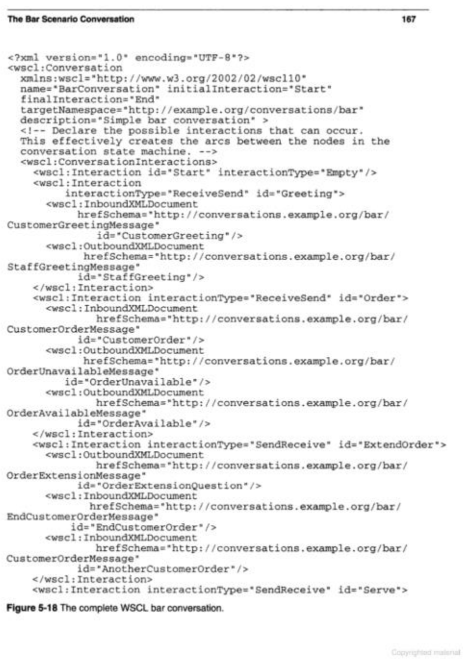
Multiplexing endpoints with empty Start and End interactions.



The Conversation element.

```
 <wscl:Conversation
xmlns:wscl="http://www.w3.org/2002/02/wscl10"
name="BarConversation" initialInteraction="Start"   finalInteraction="End"
targetNamespace="http://example.org/conversations/bar"
 description="Simple bar conversation" >
<!--The rest of the conversation omitted -->
</wscl:Conversation>
```

The Bar Scenario Conversation



```
<?xml version="1.0" encoding="UTF-8"?>
<wscl:Conversation
  xmlns:wscl="http://www.w3.org/2002/02/wscl10"
  name="BarConversation" initialInteraction="Start"
  finalInteraction="End"
  targetNamespace="http://example.org/conversations/bar"
  description="Simple bar conversation" >
  <!-- Declare the possible interactions that can occur.
  This effectively creates the arcs between the nodes in the
  conversation state machine. -->
  <wscl:ConversationInteractions>
    <wscl:Interaction id="Start" interactionType="Empty"/>
    <wscl:Interaction
         interactionType="ReceiveSend" id="Greeting">
      <wscl:InboundXMLDocument
           hrefSchema="http://conversations.example.org/bar/
CustomerGreetingMessage"
              id="CustomerGreeting"/>
      <wscl:OutboundXMLDocument
            hrefSchema="http://conversations.example.org/bar/
StaffGreetingMessage"
           id="StaffGreeting"/>
    </wscl:Interaction>
    <wscl:Interaction interactionType="ReceiveSend" id="Order">
      <wscl:InboundXMLDocument
             hrefSchema="http://conversations.example.org/bar/
CustomerOrderMessage"
           id="CustomerOrder"/>
      <wscl:OutboundXMLDocument
            hrefSchema="http://conversations.example.org/bar/
OrderUnavailableMessage"
         id="OrderUnavailable"/>
      <wscl:OutboundXMLDocument
             hrefSchema="http://conversations.example.org/bar/
OrderAvailableMessage"
           id="OrderAvailable"/>
    </wscl:Interaction>
    <wscl:Interaction interactionType="SendReceive" id="ExtendOrder">
      <wscl:OutboundXMLDocument
             hrefSchema="http://conversations.example.org/bar/
OrderExtensionMessage"
           id="OrderExtensionQuestion"/>
      <wscl:InboundXMLDocument
            hrefSchema="http://conversations.example.org/bar/
EndCustomerOrderMessage"
          id="EndCustomerOrder"/>
      <wscl:InboundXMLDocument
             hrefSchema="http://conversations.example.org/bar/
CustomerOrderMessage"
           id="AnotherCustomerOrder"/>
    </wscl:Interaction>
    <wscl:Interaction interactionType="SendReceive" id="Serve">
```

Figure 5-18 The complete WSCL bar conversation.

Relationship Between WSCL and WSDL

⇨ The three main aspects of a Web service's interface are:

- **Abstract interfaces:** The application payload (in the form of messages) being exchanged and the order in which they are exchanged (which is known as choreography).
- **Protocol bindings:** The protocols used to enable the sending and receipt of messages.
- **Services:** The concrete service implementation that provides a network accessible address for a particular protocol at which the given message set is understood—i.e., the location of an instance of the Web service.

WSDL and WSDL Complimentary Features

Aspect	**Function**	**WSDL**	**WSCL**
Abstract Interfaces	Choreography	N/A	Transition
	Messages	Operation	Interaction
Protocol Bindings		Binding	N/A
Concrete Services		Service	N/A

Where WSDL and WSCL overlap, we can map the different terminology used as shown in following Table

Mapping WSDL to WSCL

WSDL	**WSCL**
Port Type	**Conversation**
Operation • One-way • Request-response • Solicit-response • Notification	Interaction • Receive • ReceiveSend • SendReceive • Send
Input	InboundXMLDocument
Output, Fault	OutboundXMLDocument
Operation name	ID attribute of Interaction element
Operation input	ID attribute of InboundXMLDocument
Operation output	ID attribute of OutboundXMLDocument
Message	URI (which references an external schema)

⇨ The WSCL specification suggests that there are three possible approaches for combining WSDL and WSCL into a single Web service interface, depending on which stage the development of the Web service interface has reached. These are:

1. Where only abstract aspects of the WSDL interface exist, a conversational binding can be added to the WSDL interface.
2. Where existing portType declarations have been written, a cut-down WSCL description consisting of transition elements which reference existing operations can be added.
3. If a fully-formed WSDL description has already been written, a separate, full WSCL interface can be created which simply references the messages from the original WSDL document.

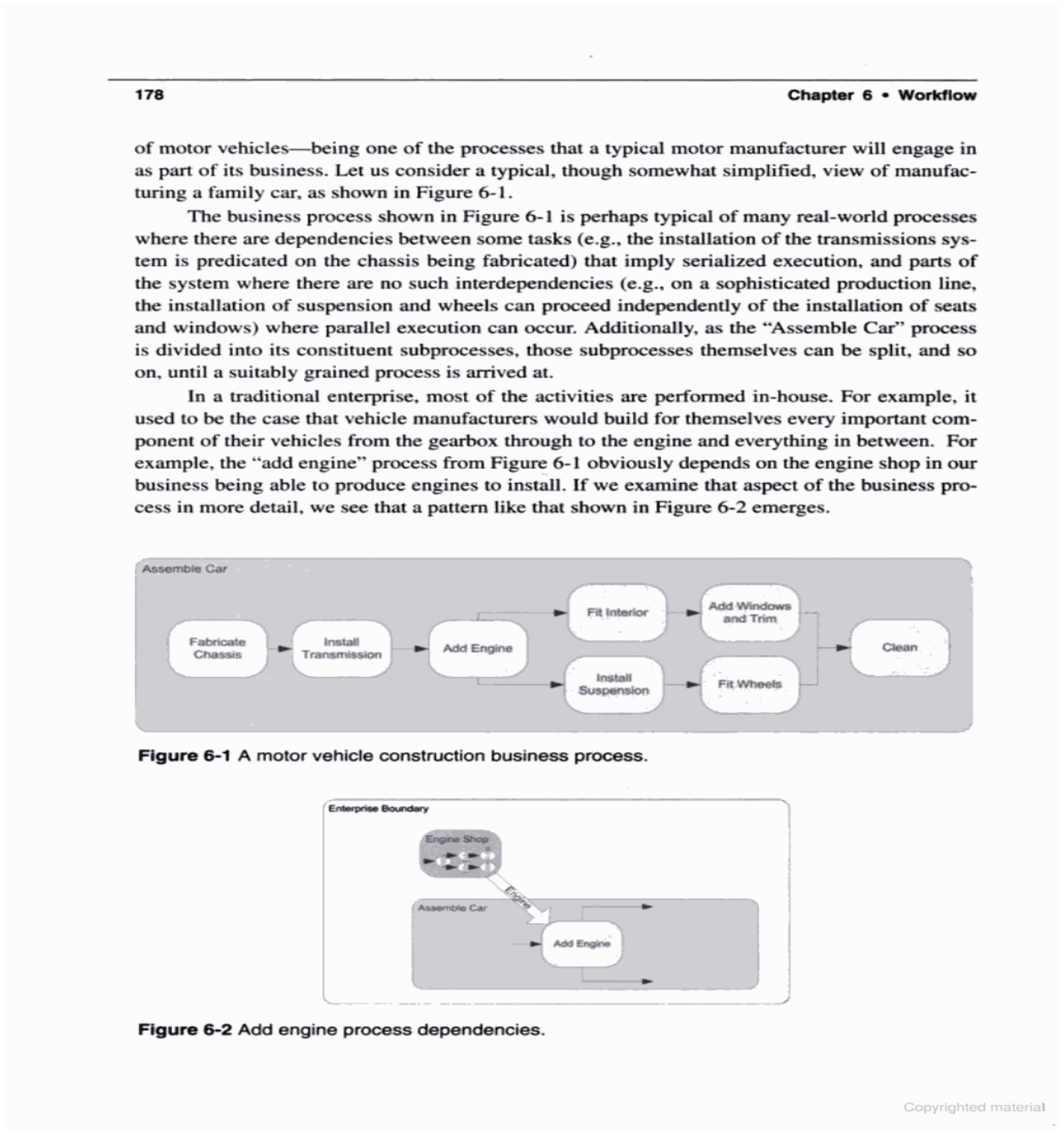
WSDL binding to WSCL conversation.

```
 <?xml version="1.0" encoding="UTF-8"?>
<definitions name="BarService"   xmlns="http://schemas.xmlsoap.org/wsdl/"
 targetNamespace=" http://example.org/wsdl/bar"
 xmlns:conv="http://example.org/conversations/bar" >
<binding name="BarServiceConversationBinding"    type="conv:BarServiceConversation">
<soap:binding style="document"/>
  <operation name="conv:Greeting">
<soap:operation soapAction="Greeting">
 </operation>
<operation name="conv:Order">
<soap:operation soapAction="Order">
</operation>   <! Other operations omitted -->
</binding>
<service name="BarService">
<port name="BarServicePort"      binding="BarServiceConversationBinding">
```

```
<soap:Address location="http://example.org/bar"/>
</port>
</service>
</definitions>
```

Workflow Business Process Management

- The model of a business as the sum of its processes is a useful abstraction from which to begin understanding a business
- For example, a supermarket is the summation of processes that buy and sell produce and a handful of other processes that keep the shop running
- Similarly, but on a grander scale, an airline consists of processes for flight sales, plane maintenance, and so on right down to the processes that ensure that passengers receive meals on board the aircraft



of motor vehicles—being one of the processes that a typical motor manufacturer will engage in as part of its business. Let us consider a typical, though somewhat simplified, view of manufacturing a family car, as shown in Figure 6-1.

The business process shown in Figure 6-1 is perhaps typical of many real-world processes where there are dependencies between some tasks (e.g., the installation of the transmissions system is predicated on the chassis being fabricated) that imply serialized execution, and parts of the system where there are no such interdependencies (e.g., on a sophisticated production line, the installation of suspension and wheels can proceed independently of the installation of seats and windows) where parallel execution can occur. Additionally, as the "Assemble Car" process is divided into its constituent subprocesses, those subprocesses themselves can be split, and so on, until a suitably grained process is arrived at.

In a traditional enterprise, most of the activities are performed in-house. For example, it used to be the case that vehicle manufacturers would build for themselves every important component of their vehicles from the gearbox through to the engine and everything in between. For example, the "add engine" process from Figure 6-1 obviously depends on the engine shop in our business being able to produce engines to install. If we examine that aspect of the business process in more detail, we see that a pattern like that shown in Figure 6-2 emerges.

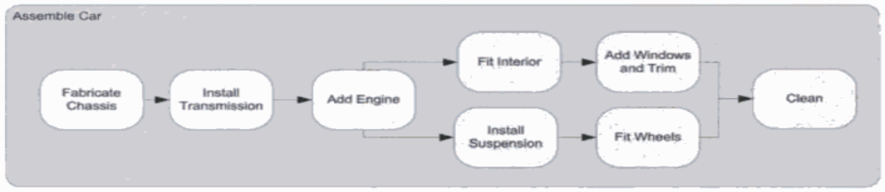


Figure 6-1 A motor vehicle construction business process.

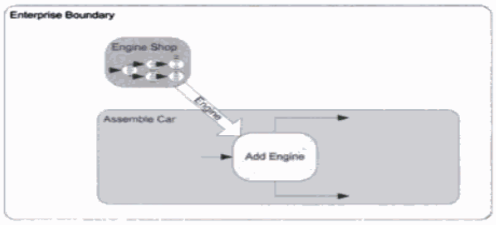


Figure 6-2 Add engine process dependencies.

While this might seem like an eminently sensible arrangement at first (especially since it decouples the internals of building engines from the internals of assembling cars), some thought reveals that in fact it hampers business agility since any change in this practice has to be capitalized up front. For instance, consider the situation where our traditional business has decided to offer a range of hybrid hydrogen fuel cell/gasoline-based cars. Under normal circumstances our business would need to find the capital and expertise to begin to produce its new engine configurations in-house, which is an expensive and labor-intensive process.

However, let us assume for argument's sake that there exist other companies whose own core competencies are not in building cars, but in building hybrid engines for cars. Given the fact that our vehicle assembly and engine construction processes are decoupled, it should be possible to simply source the new engines from a third-party supplier rather than from our own in-house stock (provided the two types of engine are, of course, physically compatible). This leads to a situation shown in Figure 6-3 where the processes of our suppliers are linked into those of the vehicle manufacturer to form a "virtual enterprise." In this arrangement, the ultimate source of the engine for a particular vehicle can either be in-house or third-party, depending on the requirements of that vehicle, without upsetting the vehicle assembly process and thus allowing the company to continue to manufacture cars and add value. Of course some of the processes within the "engine shop" may need to change a little to accommodate purchasing of complete engines (as opposed to previously purchasing raw materials or simple components) but, as a whole, the changes to the overall process are relatively confined in scope to the engine shop's systems and are (hopefully) painless.

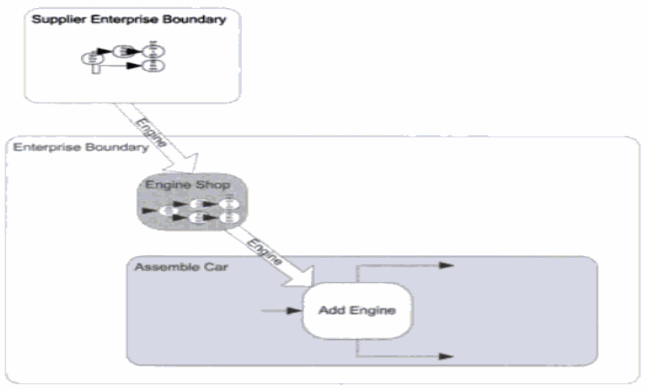


Figure 6-3 The virtual enterprise.

<u>Workflow and Workflow Management Systems</u>

Workflow management

- ⇨ is creating and optimizing the paths for data in order to complete items in a given process.
- ⇨ Workflow management includes finding redundant tasks, mapping out the workflow in an ideal state, automating the process, and identifying bottlenecks or areas for improvement.

A **workflow management system** (WMS or WfMS)

- ⇨ is a software tool designed to help streamline routine business processes for optimal efficiency. Workflow management systems involve creating a form to hold data and automating a sequential path of tasks for the data to follow until it is fully processed.
- ⇨ WfMS may also be enhanced by using existing enterprise infrastructure such as Microsoft Outlook or Office 365. A better solution would be to implement powerful workflow tools specific to building flexible workflows.
- ⇨ Tasks in workflows may be done by a human or by a system. With so many options and so many products calling themselves a workflow tool, it's hard to know what you will get.

Business Process Execution Language (BPEL)

Business Process Execution Language (BPEL)[1] defines a notation for specifying business process behavior based on Web Services. Business processes can be described in two ways:

- Executable business processes model actual behavior of a participant in a business interaction.
- Business protocols, in contrast, use process descriptions that specify the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal behavior. The process descriptions for business protocols are called abstract processes.

BPEL is used to model the behavior of both executable and abstract processes. The scope includes:

- Sequencing of process activities, especially Web Service interactions
- Correlation of messages and process instances
- Recovery behavior in case of failures and exceptional conditions
- Bilateral Web Service based relationships between process roles