11 From Web Services to Semantic Web Services

Web services will benefit from the vision of the Semantic Web. But how will these benefits arise? How are the Semantic Web and Web services related? To understand these interesting and exciting questions, it is necessary to have a solid understanding of what a Web service is and what the main components of a Web service are. Let us build a foundation in this chapter. Again, it is impossible to fit a full-length description of all the components into a single chapter; we will mainly cover the key techniques and components, which will be enough for us to start exploring Semantic Web services.

11.1 WEB SERVICE AND WEB SERVICE STANDARDS

The client-server architecture has long been the favorite choice for building distributed applications. A Web service is one such structure; it is an application (server) that provides a Web-accessible API, so that another application (client) can invoke this application programmatically.

Perhaps the fundamental issue that needs to be addressed is the communication between client and server. At present, the common agreement is that an ideal solution to this problem is to use Hypertext Transfer Protocol (HTTP) as the communication protocol. The obvious two reasons are the following:

1. HTTP is everywhere — any machine that can run a Web browser supports HTTP because a Web browser's protocol is HTTP.
2. Firewalls normally allow HTTP traffic; in other words, you can use HTTP to talk to any machine.

Clearly, these two reasons are compelling, and Web Service standards are a direct product of this solution; it is a set of standards built directly upon HTTP. By following these standards, applications can communicate with each other and achieve interoperability via the Web.

The most impressive and exciting part of the Web service standards is that if both server and client follow these standards, they can communicate with each other over HTTP regardless of their choice of platform or programming language.

Let us now take a closer look at these standards. The following three components are the key pieces of the standards.

11.1.1 Describe Your Web Service: WSDL

WSDL [45] stands for Web Service Description Language; it is an XML-based language for describing the service, including the service name, functions, and input and output parameters. In other words, it is an advertisement of the service you provide. The reason why XML is selected as the basic syntax is that XML is a pure text format; it is platform independent, can be easily parsed by any programming language, and it is fairly easy to read.

Let us see an example of how WSDL is used to describe a Web service. A Web service that takes the model of the camera as its single input parameter and returns the megapixel value as its single output will look like this:

```
double getMegaPixel(string cameraModel)
```

To ensure that there is enough information to actually invoke this service, the following items need to be included in my advertisement:

1. The location of the service; normally, the service is located at some URL. `http://www.yuchen.net/ws/getMegaPixel.asmx`, for instance, can be the location of the service.
2. The name of the service; in our case, it is `getMegaPixel`.
3. The input and output parameters; in our case, the service takes an `xsd:string` and returns an `xsd:double`.
4. The protocol to use when invoking the service and passing information between the server and client; in our case, the service is accessible using Simple Object Access Protocol (SOAP) over HTTP.

We will discuss SOAP in the next section. However, you might wonder why we need to inform the world that SOAP over HTTP has to be used to call the service, given that HTTP has already been chosen as the communication protocol. The fact is that HTTP is the wire protocol for data transmission over the Internet; however, you can use HTTP GET, HTTP POST, or SOAP to pass information back and forth between Web services and consumers. Therefore, we still need to explicitly tell the developers that this service has to be consumed using SOAP.

Now that we know what to include in the WSDL document, the next question relates to how a document can be constructed that properly advertises our service by including all the previous items.

In most cases, WSDL documents are generated automatically by the server that hosts the Web service. Because understanding WSDL documents is very important to us (you will see the reason in subsequent chapters), let us go through the process of manually generating the document.

One way to do this is to build our sample Web service using Microsoft's Visual Studio.NET. I assume that you are familiar with VS.NET, so I am not going to cover all the details. Fire up VS.NET and create a new C# Web service project named `GetMegaPixelWS`, as shown in Figure 11.1. VS.NET automatically creates a virtual directory under IIS, which is the hosting server on your local machine. Switch to the code behind, and edit the file to make it look like the one shown in List 11.1.

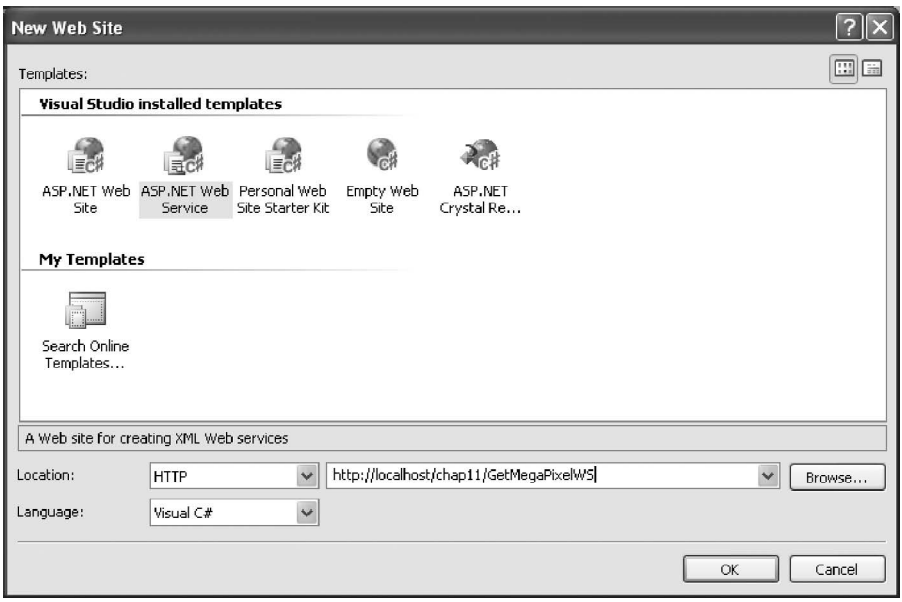

FIGURE 11.1 Creating a VS.NET Web service project.

LIST 11.1
Example of a Web Service Generated by VS.NET

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

namespace GetMegaPixelWS
{
  /// <summary>
  /// Summary description for Service1.
  /// </summary>
  public class Service1 : System.Web.Services.WebService
  {
    public Service1() { InitializeComponent(); }
    private IContainer components = null;
    private void InitializeComponent() { }
    protected override void Dispose( bool disposing )
    {
      if(disposing && components != null)
      {
        components.Dispose();
```

```
      }
      base.Dispose(disposing);
    }

    [WebMethod]
    public double getMegaPixel(string cameraModel)
    {
      return 6.0; // just testing!!
    }
  }
}
```

Note that you need to only add the `getMegaPixel` function; the rest of the code in List 11.1 is generated by VS.NET. Now run the project; VS.NET will show an autogenerated HTML page for us to test the new Web service. On this page, there is a link to the WSDL document generated by VS.NET. Click on this link, and you will see the WSDL of our simple Web service example, as shown in List 11.2.

LIST 11.2
WSDL Document for the Web Service in List 11.1

```
1:  <?xml version="1.0" encoding="utf-8"?>
2:  <definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
3:               xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
4:               xmlns:s="http://www.w3.org/2001/XMLSchema"
5:               xmlns:s0="http://tempuri.org/"
6:               xmlns:soapenc="http://schemas.xmlsoap.org/soap/
                       encoding/"
7:               xmlns:tm="http://microsoft.com/wsdl/mime/
                       textMatching/"
8:               xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
9:               targetNamespace="http://tempuri.org/"
10:              xmlns="http://schemas.xmlsoap.org/wsdl/">
11:    <types>
12:       <s:schema elementFormDefault="qualified"
                    targetNamespace="http://tempuri.org/">
13:          <s:element name="getMegaPixel">
14:             <s:complexType>
15:                <s:sequence>
16:                   <s:element minOccurs="0" maxOccurs="1"
                                 name="cameraModel" type="s:string" />
17:                </s:sequence>
18:             </s:complexType>
19:          </s:element>
20:          <s:element name="getMegaPixelResponse">
21:             <s:complexType>
22:                <s:sequence>
23:                   <s:element minOccurs="1" maxOccurs="1"
```

```
                              name="getMegaPixelResult" type="s:
                                    double" />
24:                </s:sequence>
25:             </s:complexType>
26:          </s:element>
27:       </s:schema>
28:    </types>
29:    <message name="getMegaPixelSoapIn">
30:       <part name="parameters" element="s0:getMegaPixel" />
31:    </message>
32:    <message name="getMegaPixelSoapOut">
33:       <part name="parameters" element="s0:getMegaPixelResponse" />
34:    </message>
35:    <portType name="Service1Soap">
36:       <operation name="getMegaPixel">
37:          <input message="s0:getMegaPixelSoapIn" />
38:          <output message="s0:getMegaPixelSoapOut" />
39:       </operation>
40:    </portType>
41:    <binding name="Service1Soap" type="s0:Service1Soap">
42:       <soap:binding transport="http://schemas.xmlsoap.org/soap/
               http"
                        style="document" />
43:       <operation name="getMegaPixel">
44:          <soap:operation soapAction="http://tempuri.org/
                  getMegaPixel"
                              style="document" />
45:          <input>
46:             <soap:body use="literal" />
47:          </input>
48:         <output>
49:            <soap:body use="literal" />
50:         </output>
51:       </operation>
52:    </binding>
53:    <service name="Service1">
54:       <port name="Service1Soap" binding="s0:Service1Soap">
55:          <soap:address
         location="http://localhost/chap11/GetMegaPixelWS/Service1
                  .asmx"/>
56:       </port>
57:    </service>
58: </definitions>
```

Note that there are several new terms used in the WSDL document. Generally speaking, a Web service can contain several groups of methods (operations), and each group of methods is called a `portType`. One call from the client can invoke one method. To invoke a method, the client sends an input message and gets back an output message. Each data element in a message is called a `part`. The protocol

used to invoke the service and the format of the input and output messages are together described in a `binding`. The service itself is open to the outside world via one or more `ports`. Each `port` specifies the service URL and also the `binding` to use with the `port`.

Now let us examine the generated WSDL document. The root element of a WSDL document is `<definitions>`, and everything should be defined within this root element. The `<definitions>` element normally contains quite a few namespace definitions, and in this example (lines 2 to 10), the default namespace is the WSDL namespace given by http://schema.xmlsoap.org/wsdl/.

The first element contained by the root element is the `<types>` element (lines 11 to 28). This is an XML schema (XSD) piece where you will find all the definitions of the user-defined data type as well as the built-in data types. In our case, there is no user-defined data type; all the parameters can be represented by using XSD built-in data types.

The next two elements are the `<message>` (lines 29 to 34) elements, which at first glance can be very confusing. However, a careful study will tell you what these elements are used for. The first `<message>` element represents the input message, and it has a unique name, `getMegaPixelSoapIn`, specified by its `name` attribute. As you might have guessed, input parameters should be defined within this `<message>` element. In our example, there is only one input parameter, and therefore you see only one `<part>` element defined. This `<part>` element further uses two attributes to define itself as the input parameter. The first attribute is the `name` attribute, which is used to specify the name of the input parameter. As you can see, VS.NET simply uses `parameters` as parameter name. In fact, you can use anything to represent the name. For example, to make the point clearer, we can change it to `cameraModel`. The second attribute within the `<part>` element is the `element` attribute, whose value is `s0:getMegaPixel`. This value actually serves as a link pointing to the element named `getMegaPixel`, defined in the `<types>` element at line 13. Take a look at line 13, which is the first line of the definition of the `getMegaPixel` element. This definition ends at line 19, and these lines are an XML schema piece; all it says is that the type of the `cameraModel` parameter has to be `xsd:string`.

Now, after all these details, the first `<message>` element (lines 29 to 31) can be interpreted as follows:

> The request message for this service is called `getMegaPixelSoapIn`; it has only one input parameter called `parameters`, and its type is `xsd:string`.

As we have previously mentioned, we can change the parameter name to make it more readable. In fact, we can change it to the following, and it would still be a legal WSDL `<message>` element, as shown in lines 29 to 31:

```
29:     <message name="getMegaPixelSoapIn">
30:        <part name="cameraModel" type="xsd:string"/>
31:     </message>
```

Now, it is much easier to understand the second `<message>` (lines 32 to 34) element. This element specifies the response message, and we can interpret it as follows:

The response message for this service is call `getMegaPixelSoapOut`; it has one output parameter called `parameters`, and its type is `xsd:double`.

Again, to make it more readable, we change it as follows:

```
32:     <message name="getMegaPixelSoapOut">
33:        <part name="megaPixelValue" type="xsd:double"/>
34:     </message>
```

The next element is the `<portType>` element (lines 35 to 40). It has a unique name, defined by using the `name` attribute. In our case, this name is `Service1Soap`. The `<portType>` element defines a group of methods supported by this Web service, and each such method is defined by an `<operation>` element within the `<portType>` element. Our example has only one method; therefore, you see only one `<operation>` element within the `<portType>` element.

The `<operation>` element (lines 36 to 39) has a name, `getMegaPixel`, as defined in line 36. Inside the `<operation>` element are the `<input>` and the `<output>` elements. The `<input>` element provides a pointer to the request `<message>`, and the `<output>` element provides a pointer to the response `<message>`. These pointers are implemented by using the unique names of the request and response messages. In sum, we can interpret the `<portType>` element as follows:

This Web service supports a group of methods, and this group is called `Service1Soap`. This group contains one method, and its name is `getMegaPixel`; its input message is represented by the message named `getMegaPixelSoapIn`, and its output message is represented by the message named `getMegaPixelSoapOut`.

Next is the `<binding>` element (lines 41 to line 52). The goal of this element is to show how to invoke methods within a particular `portType` (a group of methods) using a particular protocol. It has a unique name defined by its `name` attribute. In our case, this `<binding>` element has a name, `Service1Soap`. The second attribute, `type`, specifies the name of the `portType`. In our example, it is `s0:Service1Soap`. Therefore, this `<binding>` element specifies how a client should call the methods included in the `s0:Service1Soap portType`. This part of the WSDL is not much related to the Semantic Web services, and we will not get into the details at this point.

The last element in this WSDL document is the `<service>` element (lines 53 to 57). It has a `name` attribute, which specifies the name of this service, and in our example, this Web service is called `Service1`. The `<service>` element includes a `<port>` element, which provides information about how this service should be invoked. As the information needed to actually invoke this service is already encoded in the `<binding>` element, this `<port>` element just serves as a reference to the `<binding>` element, `s0:Service1Soap`. The last element within the `<service>` element is the `<soap:address>` element, which specifies the SOAP end point. In other words, the SOAP request from the client should be sent to this location. In our example, because we are hosting this service using the local IIS, you can see the URL of the service, `http://localhost/chap11/GetMegaPixelWS/Service1.asmx`.

We have now finished examining the generated WSDL document, which carefully provides all the information that might be needed by a client who wishes to

invoke the service. Here, we have used quite a few pages to describe a WSDL document; later, we will make changes to this document to convert the underlying Web service into a Semantic Web service. So you really need to have a solid understanding of WSDL.

11.1.2 Exchange Data Freely: SOAP

We have mentioned SOAP [46] several times in the previous section. It defines a standard protocol specifying how one application should communicate and exchange data with another application over the Internet. The significance of SOAP lies in its relationship with HTTP. Let us take a closer look at it.

There are millions of computers on the Internet, and the Internet itself provides the basic networking to interconnect all these computers. This basic network connectivity is supported by TCP/IP. However, TCP/IP is at such a low level that if a given computer wants to talk to another computer, it has to write its own higher-level communication code. This is certainly inefficient and nonstandard. Therefore, some kind of higher-level communication support is needed to make life easier.

HTTP was designed as a solution to this problem. It is a higher-level (application-level) protocol designed for communication between two machines on the Internet. It is said to be at a higher level because HTTP is built on top of TCP/IP, and it successfully eliminates much needless reinventing of wheels. However, HTTP does not constitute a complete solution. It is designed only for use between a Web browser and a Web server. In other words, under HTTP, communication between computers on the Internet mainly occurs in the form of browsing the Web.

What happens when two computers need to exchange data that are much more complex than the simple GET, POST, or PUT? This need is obvious in the world of Web services, where different kinds of application-level data have to be exchanged freely between machines.

SOAP was invented just for this purpose. If one application on a given computer sends data by following the SOAP protocol, another application on a different computer would be able to make use of the data without any problem, regardless of the programming language used by each one of these two applications or the platforms the applications are running on.

Now, what is the relationship between SOAP and HTTP? Normally, SOAP is layered over HTTP. This is not necessary, but because most firewalls allow only HTTP traffic, it is a very simple and practical solution. In other words, most SOAP messages will be sent or received as part of the HTTP request or response.

As a summary, for any two applications running on two different machines, data exchange over the Internet can be implemented by using the SOAP protocol. Furthermore, because a SOAP message itself is an XML message, it does not matter whether these two applications are developed using the same programming language or the two computers are running on different platforms; communication can always be achieved.

By now, you should be able to see the role SOAP plays in the world of Web services. A given Web service consumer (client) and the Web service provider (server) are two applications, normally running on two different machines over the

Internet, and these two applications need to exchange data back and forth; SOAP is used to make this communication possible.

The good news is that the SOAP messages for Web service requests and responses are normally generated automatically based on the WSDL document, and it is very rare you would need to manually change it; it is the lower-level layer that makes the data exchange successful.

SOAP does not play much of a role either in the Semantic Web services world, and it is therefore not of much use to examine SOAP messages in detail. Just for our viewing pleasure though, List 11.3 and List 11.4 show the SOAP request and response, respectively, for the simple Web service we coded in the previous section (note that the italic words in List 11.3 and List 11.4 are placeholders; they need to be replaced by the real values when the service is being consumed).

LIST 11.3
SOAP Message for the Request

```
POST /chap11/GetMegaPixelWS/Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/getMegaPixel"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap=
"http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getMegaPixel xmlns="http://tempuri.org/">
      <cameraModel>string</cameraModel>
    </getMegaPixel>
  </soap:Body>
</soap:Envelope>
```

LIST 11.4
SOAP Response for the Request Shown in List 11.3

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://
schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getMegaPixelResponse xmlns="http://tempuri.org/">
```

```
      <getMegaPixelResult>double</getMegaPixelResult>
    </getMegaPixelResponse>
  </soap:Body>
</soap:Envelope>
```

11.1.3 Typical Activity Flow for Web Services

So far, we have learned that a Web service is described by using WSDL; SOAP messages are used to exchange data between the client and the server; and both the WSDL document and the SOAP messages use XML syntax. Now, to conclude this section, we give the overall activity flow for traditional Web services, so we understand how these pieces are connected to work together:

1. A Web service is created by using some programming language.
2. This Web service is exposed by publishing its WSDL document, so that potential clients can access it. Normally, the WSDL document is generated by using a provided tool or by the built-in support of the development environment.
3. A Web server will host this Web service (IIS in our example) by listening to the HTTP traffic.
4. A client application (probably written in another programming language) accesses the WSDL document, and a SOAP request message is generated based on the WSDL document.
5. The Web server receives the SOAP request as part of an HTTP POST request, and it forwards this request to a Web service request handler (a system-level application that is always running).
6. The Web service request handler parses the SOAP message, invokes the right Web service, and also creates the SOAP response. It finally sends the response to the Web server.
7. The Web server formulates an HTTP response, which includes the SOAP response message, and sends it back to the client.

11.2 FROM WEB SERVICES TO SEMANTIC WEB SERVICES

11.2.1 UDDI: A Registry of Web Services

In the previous section, we discussed the process of creating, publishing, and consuming a Web service. There is only one problem in consuming a Web service: a client has to be able to discover it first. How can the client discover the service?

For instance, suppose I need to find a Web service that is able to tell me the temperature at a given major airport in the U.S.A. Ideally, this service would accept an airport code as input (each major airport has an airport code; for instance, Atlanta's main airport has the code "ATL"), and return the current temperature as output. This seems to be a very trivial service; however, considering the dynamic nature of temperature (it is changing all the time), it turns out to be a very important service.

The problem is that we do not even know whether such a service exists at all, so our first step is to try to discover it. The good news is that there is another major piece in the Web service arena that we have not yet talked about, and this piece is specifically designed for discovering Web services; it is called UDDI [47].

UDDI stands for Universal Description Discovery and Integration; its main function is to provide support for publishing and finding service descriptions. It has a directory structure where businesses can register and search for Web services. In this section, we will study UDDI to understand its support for Web service publishing and discovering.

Before we discuss the details of UDDI, let us first modify the Web service activity flow description to complete the picture:

1. A Web service is created using some programming language.
2. The Web service is described by a WSDL document (normally generated by using a provided tool or by the built-in support of the development environment).
3. The service provider publishes the Web service into the UDDI repository.
4. A Web server hosts this Web service by listening to HTTP traffic.
5. A client application (probably written in another programming language) searches the UDDI registry and discovers this service.
6. The client accesses the WSDL document, and a SOAP request message is generated based on the WSDL document.
7. The Web server receives the SOAP request as part of a HTTP POST request, and it forwards this request to a Web service request handler (a system-level application that is always running).
8. The Web service request handler parses the SOAP message, invokes the right Web service, and also creates the SOAP response. It finally sends the response to the Web server.
9. The Web server formulates a HTTP response, which includes the SOAP response, message and sends it back to the client.

Now that we have a better idea of the typical activity flow of a Web service, we can take a closer look at UDDI. There are four main data types in a UDDI directory: `businessEntity`, `businessService`, `bindingTemplate`, and `tModel` (in fact, there is another data type called `publisherAssertion`; it is not relevant for our purposes, so let us not worry about it for now). Figure 11.2 shows these main types.

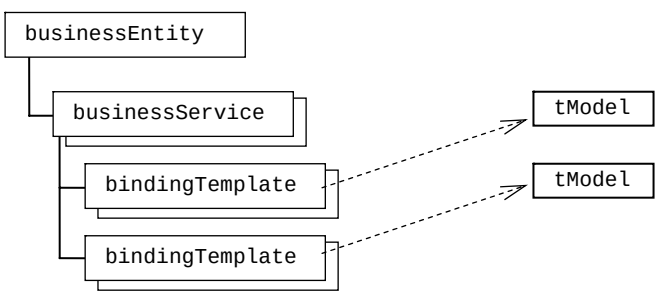

FIGURE 11.2 Basic UDDI data types.

The businessEntity data structure is used to represent a business or a service provider. If we want to publish a service with UDDI, we have to publish a new businessEntity first to represent ourselves as a business unit. List 11.5 is the XML schema for businessEntity. Therefore, an instance of businessEntity can have attributes or elements summarized in Table 11.1.

LIST 11.5
XML Schema for businessEntity

```
<element name="businessEntity" type="uddi:businessEntity" />
<complexType name="businessEntity">
  <sequence>
     <element ref="uddi:discoveryURLs" minOccurs="0" />
     <element ref="uddi:name" maxOccurs="unbounded" />
     <element ref="uddi:description" minOccurs="0" maxOccurs=
                 "unbounded"/>
     <element ref="uddi:contacts" minOccurs="0" />
     <element ref="uddi:businessServices" minOccurs="0" />
     <element ref="uddi:identifierBag" minOccurs="0" />
     <element ref="uddi:categoryBag" minOccurs="0" />
  </sequence>
  <attribute name="businessKey" type="uddi:businessKey" use=
                 "required" />
  <attribute name="operator" type="string" use="optional" />
  <attribute name="authorizedName" type="string" use="optional" />
</complexType>
```

TABLE 11.1
Attributes and Elements for businessEntity

Attribute/Element	Meaning
businessKey	Attribute; the unique identifier of this instance
authorizedName	Attribute; name of the individual who published this instance
operator	Attribute; name of the UDDI registry site operator who manages the master copy of the businessEntity data
discoveryURLs	Element; a list of URLs pointing to other possible service discovery mechanisms
name	Element; human-readable names (possibly in different languages) of the businessEntity.
description	Element; human-readable description of the businessEntity
contacts	Element; list of contact information
businessService	Element; list of one or more logical business service description structures (more below)
identifierBag	Element; list of (name, value) pairs used to record identifiers for a businessEntity (more below)
categoryBag	Element; list of (name, value) pairs used to tag a businessEntity with specific taxonomy information (more below)

Most of these attributes and elements are fairly intuitive and straightforward. The first is the `businessKey` attribute. It is the key attribute used to uniquely identify the business that offers the service. This key is called UUID (Universally Unique Identifier), and we do have tools to generate such keys. It is normally assigned to the `businessEntity` by the `operator` when it is published; so we do not have to worry about where to get it from.

The `discoveryURL` element (contained in the `discoveryURLs` element) could be a little confusing. A `businessEntity` may have one or more of these URLs, which point to so-called discovery documents. However, these documents normally have nothing to do with Web services; they are simply documents that provide extra information about the business itself. For example, one such discovery URL could be the homepage of the business.

The `identifierBag` and `categoryBag` elements could be confusing, too. They are mainly used to add identification and categorization information to a `businessEntity`. To fully understand them, we need to know another major data type, `tModel`. So, let us discuss these two elements after we know more about `tModels`.

Lastly, a `businessEntity` may contain zero or more `businessService` structures. A given `businessService` structure normally represents a specific service provided by the `businessEntity`; it is the entrance to the real service that we care about. List 11.6 is the XML schema for `businessService` structure, and its attributes and elements are summarized in Table 11.2.

LIST 11.6
XML Schema for `businessService`

```
<element name="businessService" type="uddi:businessService" />
<complexType name="businessService">
  <sequence>
    <element ref="uddi:name" minOccurs="0" maxOccurs="unbounded" />
    <element ref="uddi:description" minOccurs="0" maxOccurs=
                "unbounded" />
    <element ref="uddi:bindingTemplates" minOccurs="0" />
    <element ref="uddi:categoryBag" minOccurs="0" />
  </sequence>
  <attribute name="serviceKey" type="uddi:serviceKey" use="required" />
  <attribute name="businessKey" type="uddi:businessKey" use=
                 "optional" />
</complexType>
```

Note that the `businessKey` attribute is tagged as `optional` in the schema. However, in the real world, it is normally necessary to have this key point back to the `serviceEntity` instance that offers this service. In other words, every `businessService` instance should be a child of a single `serviceEntity` instance. Another important attribute is the `serviceKey` attribute, which is used to uniquely identify this service. It is certainly required and, again, is automatically assigned by the UDDI registry (another long UUID number). The confusing element in this

TABLE 11.2
Attributes and Elements for `businessService`

Attribute/Element	Meaning
`businessKey`	Attribute; a reference to the UUID key of the containing `businessEntity` instance
`serviceKey`	Attribute; unique businessService key to identify this service
`name`	Element; human-readable name of this service
`description`	Element; human-readable description of this service
`bindingTemplates`	Element; a reference to the technical details of the service
`categoryBag`	Element; pairs to categorize the service (name, value)

schema could again be the `categoryBag` element. We will discuss more about this later, but for now just understand that it is used to categorize the underlying service, chiefly to facilitate search within the UDDI repository.

The `bindingTemplate` element links this service to its own technical descriptions. Every given `businessService` instance contains zero or more `bindingTemplate` structures. One such structure defines technical information, such as the service's interface and end point URL of the service. List 11.7 is the XML schema for the `bindingTemplate`. Table 11.3 is a summary of the attributes and elements of the `bindingTemplate` structure.

LIST 11.7
XML Schema for `bindingTemplate`

```
<element name="bindingTemplate" type="uddi:bindingTemplate" />
<complexType name="bindingTemplate">
  <sequence>
    <element ref="uddi:description" minOccurs="0" maxOccurs=
                "unbounded" />
    <choice>
      <element ref="uddi:accessPoint" />
      <element ref="uddi:hostingRedirector" />
    </choice>
    <element ref="uddi:tModelInstanceDetails" />
  </sequence>
  <attribute name="serviceKey" type="uddi:serviceKey" use="optional" />
  <attribute name="bindingKey" type="uddi:bindingKey" use="required" />
</complexType>
```

Again, each `bindingTemplate` is uniquely identified by a system-generated UUID, and this value is stored in the `bindingKey` attribute. Although the `serviceKey` attribute is tagged as optional, it is normally used to point back to its parent `businessService` instance.

TABLE 11.3
Attributes and Elements for `bindingTemplate`

Attribute/Element	Meaning
`bindingKey`	Attribute; unique key of the `bindingTemplate` instance
`serviceKey`	Attribute; a reference pointing to the `businessService` parent instance
`description`	Element; human-readable descriptions
`accessPoint`	Element; entry point address of this service (more discussion below)
`hostingRedirector`	Element; this is required if `accesPoint` is not provided; in this case it points to a remote `bindingTemplate` (more discussion below)
`tModelInstanceDetails`	Required container to hold the `tModel` details

Note the `<choice>` element in the preceding schema. This states that each `bindingTemplate` instance must contain either an `accessPoint` or a `hostingRedirector` element. An `accessPoint` element contains the URL where this service can be accessed. If the `accessPoint` element is not present in the `bindingTemplate` instance, the `hostingRedirector` element must be present, pointing to another `bindingTemplate`. There are cases in which this feature is useful, but for our purposes, let us not dive deep into it for now.

The last element is `tModelInstanceDetails`. This seems to be confusing at first glance, but let us think about it. The `bindingTemplate` is where all the technical details of the underlying service are described. The details given here are twofold: the access URL of the service and all the other details. To indicate the access URL, `accessPoint` and `hostingRedirector` elements are provided by the `bindingTemplate`; for all the other technical details of the service, the `tModelInstanceDetails` element is the answer. It can be thought of as a container that holds all the technical information (`tModels`) of a given service. tModel is one of the major data structures in UDDI. List 11.8 is the XML schema for `tModel`.

LIST 11.8
XML Schema for `tModel`

```
<element name="tModel" type="uddi:tModel" />
<complexType name="tModel">
  <sequence>
    <element ref="uddi:name" />
    <element ref="uddi:description" minOccurs="0" maxOccurs=
                "unbounded" />
    <element ref="uddi:overviewDoc" minOccurs="0" />
    <element ref="uddi:identifierBag" minOccurs="0" />
    <element ref="uddi:categoryBag" minOccurs="0" />
  </sequence>
```

```
  <attribute name="tModelKey" type="uddi:tModelKey" use="required" />
  <attribute name="operator" type="string" use="optional" />
  <attribute name="authorizedName" type="string" use="optional" />
</complexType>
```

Note that `tModel`s are first-class data structures in UDDI, meaning that they are not contained by any entity, but can show up in any entity. Each `tModel` is uniquely identified by its `tModelKey`, again an automatically generated UUID string. A `tModel` also has a `name` and optional `description` elements. Its `overviewURL` element often points to a document that describes the service interface; it is perfectly legal to have this element point to other documents.

To understand exactly what `tModel` can do for us, let us study some examples. Recall the camera Web service example presented in the previous section. It takes a string representing the camera model as input and returns a megapixel number of the given camera. Based on the flowchart of Web services, we know that for potential clients to use this service we need to publish it into the UDDI first.

In fact, to publish it into UDDI as a Web service entry, we need to create a service type first. A service type is simply an interface. The goal of this interface (no implementation details are included) is to tell the world that a particular Web service is being offered, and that it can be discovered in the UDDI registry. This service type is registered with the UDDI repository by creating a `tModel` data structure to represent it. List 11.9 shows what this `tModel` looks like (again, there are tools you can use to create and register a given `tModel`). This `tModel`, representing a service type (interface), declares the following:

> This is a tModel representing a service type; this service type is called `getMegaPixel`, and the URL of its WSDL document is at the following address:
>
> http://localhost/chap11/GetMegaPixelWS/Service1.wsdl

LIST 11.9
The `tModel` Representing the Interface of the Example Web Service

```
1: <tModel tModelKey="uuid:5DD52389-B1A4-4fe7-B131-0F8EF73DD175">
2:     <name>getMegaPixel</name>
3:     <description>interface for camera Web service</description>
4:     <overviewDoc>
5:        <description xml:lang="en">URL of WSDL document</description>
6:        <overviewURL>
             http://localhost/chap11/GetMegaPixelWS/Service1.wsdl
7:        </overviewURL>
8:     </overviewDoc>
9: </tModel>
```

Now that we have this service type created and registered with UDDI, and given that we do offer a service conforming to this service type, we can go ahead and

create a UDDI entry to represent our Web service; we will reference this `tModel` in our `bindingTemplate`. The final `businessEntity` is given in List 11.10.

This is what our camera Web service looks like inside the UDDI registry, and it includes everything we have learned about UDDI up to this point. List 11.10 first

LIST 11.10
The `businessEntity` Representing the Example Web Service

```
1: <businessEntity
      businessKey="uuid:AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAAAA"
      operator="someOperatorName"
      authorizedName="somePeronsName">
2:   <name>someCompanyName</name>
3:   <discoveryURLs>
4:     <discoveryURL useType="businessEntity">
5:        http://www.someWebSite.com/someDiscoveryLink
6:     </dicoveryURL>
7:   </discoveryURLs>
8:   <businessServices>
9:     <businessService
         serviceKey="uuid:BBBBBBBB-BBBB-BBBB-BBBB-BBBBBBBBBBBB"
         businessKey="uuid:AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAAAA">
10:      <name>getMegaPixel</name>
11:      <description>returns the mega-pixel for a model</description>
12:      <bindingTemplates>
13:        <bindingTemplate
            bindingKey="uuid:CCCCCCCC-CCCC-CCCC-CCCC-CCCCCCCCCCCC"
            serviceKey="uuid:BBBBBBBB-BBBB-BBBB-BBBB-BBBBBBBBBBBB">
14:          <accessPoint URLType="http">
              http://localhost/chap11/GetMegaPixelWS/Service1.asmx
            </accessPoint>
15:          <tModelInstanceDetails>
16:            <tModelInstanceInfo
                tModelKey="uuid:5DD52389-B1A4-4fe7-B131-0F8EF73DD175">
17:              <instanceDetails>
18:                <overviewDoc>
19:                  <overviewURL>
                     http://localhost/chap11/GetMegaPixelWS/Service1
                            .wsdl
20:                  </overviewURL>
21:                </overviewDoc>
22:              </instanceDetails>
23:            </tModelInstanceInfo>
24:          </tModelInstanceDetails>
25:        </bindingTemplate>
26:      </bindingTemplates>
27:   </businessService>
28:  </businessServices>
29:</businessEntity>
```

describes our business very briefly (lines 1 to 7) and then specifies one service (lines 9 to 16) provided by the business. This service has a specific interface described by the `tModel`, with a key value given by `uuid:5DD52389-B1A4-4fe7-B131-0F8EF73DD175` (line 16). Note that I have used `uuid:AAAA...`, `uuid:BBBB...`, and `uuid:CCCC...` (lines 9 and 13) to represent the UUIDs for readability, and it is also easier for you to see the cross-reference of UUIDs inside the `businessEntity` instance.

Up to this point, we have learned how UDDI is used to register a given Web service, and we have seen at least one possible use of `tModels` to facilitate this registration process. Remember, the main purpose of UDDI is to help Web service consumers find the Web services they want. Now, with our newly acquired knowledge of UDDI, let us figure out how exactly it can help us find a desired Web service.

11.2.2 Using UDDI to Discover Web Services

Suppose a client wants to find a service that takes a camera model as input and returns a value representing the megapixel number of the given model. Without knowing whether such a service exists, the client decides to search within UDDI to find some clue.

However, it is extremely difficult to do the search; to find the service, you need to know one of the following:

- The `tModelKey` of the interface the camera service implements.
- The service name, `getMegaPixel`.
- One of these keys: `businessKey`, `serviceKey`, or `bindingKey`.
- The name of the WSDL document or its location.
- The company that has developed this service.

Obviously, the client does not have any of this information. Therefore, searching in UDDI for a totally new service is impossible. However, when you do know that a given service exists, and you also have some information about it (for example, which company developed it), the service will be easier to find.

The UDDI designers are aware of the search problems, especially the difficulty when a client wants to use UDDI to search for a new service about which he or she knows nothing. To improve the search ability, UDDI has included two types of additional information, namely, identification and categorization information, represented by two more constructs, `identifierBag` and `categoryBag`, respectively. In the next two sections, we will look at these two constructs to see how they can improve the search function in UDDI.

11.2.2.1 Adding Categorization Information to the Service Type

The first idea is to add some categorization information into the service type (interface) `tModel`. Although knowing its `tModelKey` is impossible (we do not even know whether this interface exists), it is still possible to find it by conducting a search based on categorization. For example, for the interface `tModel` we created, we can specify the category for this interface as “online information service.” If a client searches this category, our interface `tModel` will show up as the result.

It is certainly not a good idea to let every developer come up with his or her own categorization system, thereby producing a number of different categorization systems and defeating the purpose of having such a system. Therefore, some kind of standardized categorization is needed.

UDDI does provide several classification schemas for this purpose. One such schema is the classification of interface `tModel`s. Recall that we created an interface `tModel` in the previous section. More specifically, this `tModel` represents an interface for a Web service that is further described using a WSDL document. There are certainly many other interface `tModel`s representing totally different interfaces. Therefore, one possible categorization is to classify all the interface `tModel`s. This classification system is called `uddi-org:types`, and it includes quite a few different type values, such as `wsdlSpec`, `xmlSpec`, and `protocol`. Visit `http://www.uddi.org/taxonomies/Core_Taxonomy_OverviewDoc.htm` for a complete list.

But before such a classification can be used to find a service, one problem still needs to be solved: how do you represent this categorization system in UDDI? The answer is, use `tModel`. More precisely, to represent this classification in UDDI, a special `tModel` is created and preregistered (canonical) in UDDI. Such a preregistered `tModel` has the following characteristics:

- It is supported by any given UDDI registry.
- It always uses the same key (this key is `UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4`).

List 11.11 shows how to add this categorization information into our interface `tModel`. Now, this enhanced `tModel` tells us the following:

> This is a `tModel` representing a service type; this service type is called `getMegaPixel`, and the URL of its WSDL document is at the following address:
>
> http://localhost/chap11/GetMegaPixelWS/Service1.wsdl
>
> Also, this interface is categorized as of type `wsdlSepc` (specification for Web service described in WSDL) by using the `uddi-org:types` categorization system. This categorization system is represented by a `tModel` whose `tModelKey` is given by this string: `UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4` (line 11).

LIST 11.11
Adding `wsdlSpec` Categorization Information into the Interface `tModel`

```
1: <tModel tModelKey="uuid:5DD52389-B1A4-4fe7-B131-0F8EF73DD175">
2:    <name>getMegaPixel</name>
3:    <description>interface for camera Web service</description>
4:    <overviewDoc>
5:       <description xml:lang="en">URL of WSDL document</description>
6:       <overviewURL>
            http://localhost/chap11/GetMegaPixelWS/Service1.wsdl
7:       </overviewURL>
8:    </overviewDoc>
```

```
9:    <categoryBag>
10:      <keyedReference
11:          tModelKey="UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4"
12:          keyName="specification for a Web service described in
                    WSDL"
13:          keyValue="wsdlSpec"/>
14:   </categoryBag>
```

Note the syntax of adding categorization information; you have to use the `keyedReference` element in a `categoryBag`. Remember, we did not discuss `categoryBag` in the previous section. Now you see how to use it. In general, you use the `categoryBag` element to add categorization or classification information to an instance of a given UDDI data structure; the purpose is to ensure that a search based on this categorization information will locate this instance successfully.

For example, `tModel` is a UDDI data structure, and by reading its schema (List 11.8) you know it is legal to add `categoryBag` element into the data structure. Therefore, we have added some categorization information into the interface `tModel` we created, by using the `categoryBag` element, as shown in List 11.11.

In fact, if you read the XML schema for the `categoryBag` data structure, you will realize that you can add as much categorization information into the `categoryBag` structure as you want. Let us then use another popular categorization schema called NAICS1997[48] to add more classification information into our interface `tModel`.

NAICS1997 stands for North American Industry Classification System 1997 release. It provides a set of classification codes to identify a category of a specific service. Again, this classification is represented by a `tModel`, whose name is `ntis-gov.naics:1997`, with `uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2` as its `tModelKey`.

To use NAICS1997 to classify our interface `tModel`, we first need to identify the business type of the Web service this interface represents. One way to find a good business classification fit is to visit NAICS's official Web site to look for a code. You can find these codes at `http://www.census.gov/epcd/www/naics.html`. Figure 11.3 shows some of the codes.

An "Online Information Service" sounds like a good fit to classify the Web service we are providing, so let us add this to our interface `tModel`. The result is shown in List 11.12.

Now, we have added two different classifications to our simple interface `tModel` (lines 9 to 18). The first one states that the interface `tModel` has a type `wsdlSpec`, and therefore represents a Web service described by a WSDL file; the second one states that according to the NACIS1997 categorization schema, this service is a service with value `514191` (representing online information services).

The final result is a better search mechanism: a client searching for the interface `tModels`, whose type is `wsdlSpec`, will find us, and a client searching for "online information services" will also find us. Remember, there is just no way to search our interface `tModel` without the added categorization information. This is indeed quite a significant improvement.

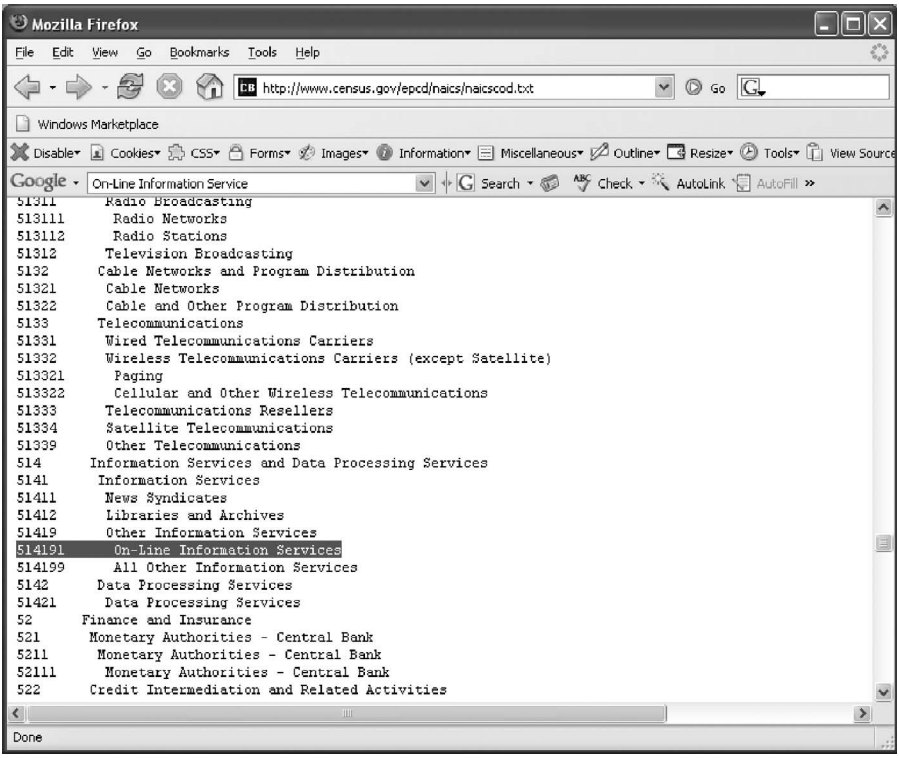

FIGURE 11.3 NAICS code (1997 release).

LIST 11.12
Adding NAICS Categorization Information into the Interface `tModel`

```
1: <tModel tModelKey="uuid:5DD52389-B1A4-4fe7-B131-0F8EF73DD175">
2:     <name>getMegaPixel</name>
3:     <description>interface for camera Web service</description>
4:     <overviewDoc>
5:        <description xml:lang="en">URL of WSDL document</description>
6:        <overviewURL>
             http://localhost/chap11/GetMegaPixelWS/Service1.wsdl
7:        </overviewURL>
8:     </overviewDoc>
9:     <categoryBag>
10:       <keyedReference
11:           tModelKey="UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4"
12:           keyName="specification for a Web service described in
                  WSDL"
13:           keyValue="wsdlSpec"/>
14:       <keyedReference
15:           tModelKey="UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
```

```
16:          keyName="On-Line Information Services"
17:          keyValue="514191"/>
18:   </categoryBag>
19: </tModel>
```

In fact, further improvement is possible. Recall that we mentioned that you can add categorization information into the `categoryBag` element of any UDDI data structure as long as that data structure allows you to do so. In fact, `businessEntity` is another UDDI data structure that contains an optional `categoryBag` element; therefore, we can add categorization information to it to make a given `businessEntity` instance more searchable.

For example, we can modify our previous `businessEntity` description (List 11.10) to include some categorization information also, as shown in List 11.13. Again, the purpose of adding categorization information into our `businessEntity` description is to facilitate search in UDDI; a client would be able to find us as one of the organizations that offer online information services.

LIST 11.13
The `businessEntity` with Categorization Information

```
1: <businessEntity
      businessKey="uuid:AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAAAA"
      operator="someOperatorName"
      authorizedName="somePeronsName">
2:    <name>someCompanyName</name>
3:    <discoveryURLs>
       ... ...
7:    </discoveryURLs>
8:    <businessServices>
       ... ...
28:   </businessServices>
29:   <categoryBag>
30:      <keyedReference
31:       tModelKey="UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
32:       keyName="On-Line Information Services"
33:       keyValue="514191"/>
34:   </categoryBag>
35:</businessEntity>
```

You can use other categorization schemas as well, but the procedure is always the same, as described earlier. Check out the following Web site for more categorization schemas that you might want to use:

`http://www.uddi.org/taxonomies/Core_Taxonomy_OverviewDoc.htm`

11.2.2.2 Adding Identification Information to the Service Type

Now you have seen how categorization information is added to help the search within the UDDI registry. Besides this categorization method, a service publisher can also help the search engine within UDDI to quickly locate relevant UDDI data structures by adding well-known identifiers to them. For example, to uniquely identify a business, we can use the federal tax ID. This additional information is the identification information.

Adding well-known identifiers to UDDI data structures is implemented by using `identifierBag`; it has exactly the same look and feel as `categoryBag`, and the only difference between these two is that `identifierBag` contains identification information whereas `categoryBag` contains categorization information.

As you might have guessed, there are well-known identifier schemas you can use, which are preregistered with UDDI by using `tModel`s. For instance, the popular D-U-N-S Number identifier system is represented by a `tModel` named `dnb-com:D-U-N-S` with a tModelKey given by `uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823`. List 11.14 is one example of using this system to identify a given business. For more information about DUNS, visit `http://www.dnb.com`.

LIST 11.14
Using `identifierBag` to Add Identification Information

```
<businessEntity businessKey=...
...
 <identifierBag>
  <keyedReference keyName="IBM Corporation"
   keyValue="00-136-8083"
   tModelKey = "uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823"/>
...
 </identifierBag>
...
</businessEntity>
```

There are certainly other identification systems for you to use. Again, they are used as shown in List 11.14.

Now we have discussed UDDI data structures, and we have also seen both categorization and identification schemas; it is time to move on to answer the most exciting questions in this chapter: How is all this related to the Semantic Web? What is the Semantic Web service?

11.2.3 The Need for Semantic Web Services

Recall that in Chapter 1 of this book we talked about the common uses of the Internet; we said that one of these uses is integration, and we also mentioned several examples

of integration. Now we should realize that these examples are use cases of Web services, which are a special form of integration. This kind of integration is of great interest to today's business organizations, because they provide a much more efficient and reliable way to exchange data, reuse functionalities and, finally, conduct business.

Having said this, what is the expectation of the real world regarding these Web services? We have also discussed this in Chapter 1, but let us formally summarize our findings here:

1. Automatic discovery of Web services: Finding the desired service can be hard, especially when the service requester does not know of the existence of the service provided; the requester's only hope would be that someone might have provided that requested service online. However, to make Web services a real success, a way to discover the requested service should be provided; also, it has to be discovered automatically, with great accuracy and efficiency.
2. Automatic invocation of the service: After the requested service has been discovered, the software agent should be able to invoke the service automatically. The benefit is obvious: with no human intervention and delay, you can conduct your business on a larger scale with much better efficiency. Also, in many cases, this automatic invocation is simply a must; some applications have to run continuously without any interruption.
3. Automatic composition of the necessary services: Quite often, a specific business need requires several Web services to work together. For instance, a replenishment of inventory will involve querying the prices from different vendors (calling the Web services provided by these vendors), comparing the prices (you can do this locally), and placing the orders (this is another Web service provided by the vendor from whom you decide to buy). Clearly, a software agent should be able to find all the necessary services and invoke them in the correct order to accomplish the business goal.
4. Automatic monitoring of the execution process: Clearly, if all the preceding processes are automatic, then how do we know whether the requested service has been found and executed successfully and correctly? There has to be some mechanism to detect and report possible failures, if any.

These are the expectations the real world has regarding Web services. Now that we have learned the main components of Web services, including WSDL, SOAP, and UDDI, how far are we from these expectations?

This might be a good time to invite you to be the judge. Actually, even manually finding the requested service is very difficult, let alone automatic invocation, composition, and monitoring. In fact, you have already learned that in the world of Web services, UDDI is provided as the main vehicle to find the requested service. However, using UDDI to find the requested Web service is extremely hard, if not impossible, for the following reasons:

- UDDI's search engine is a keyword-based search engine; in other words, you either find nothing, or you find too many.

- A variety of categorization and identification schemas have been used in various data types in UDDI to facilitate the discovery process. However, this requires a certain amount of familiarity with all these classification and identification schemas on the part of the potential clients (developers). Even assuming this is not a problem, there is still the possibility that different parties may categorize the same service differently, which could defeat the very purpose of having these categorization and identification schemas.
- Besides the preceding points, UDDI does not offer any other way of searching for the requested services.

The final result is that using UDDI to search for services is really a trial-and-error process; there is no guarantee that the manual process will succeed, and automatic discovery is impossible.

But this does not mean that UDDI has failed in its mission; if you already have some information regarding a given Web service, such as the service interface `tModelKey` or the key value of the `businessEntity` that has developed it, you can find more detailed information about it using UDDI. For instance, the search result from UDDI will let you know where you can access the corresponding WSDL document, which provides enough information for you to invoke the service.

Now it looks as if we are stuck regarding even the very first expectation: how to find the requested service automatically. Without an automatic discovery process, the other goals — automatic invocation, automatic composition, and automatic monitoring — are impossible to realize. For this reason (among others), the rest of this book will mainly concentrate on the discovery of Web services.

The difficulty of finding the requested Web service on the Web inevitably remains as one of the main motivations of the Semantic Web vision. Searching information on the Web is so difficult that people started to think about adding semantics to Web pages to make the search more relevant and efficient, and as you have seen, this does open up a whole new world of great expectations, and some results are already very encouraging.

Realizing that automatic discovery of Web services is just another searching activity, researchers and developers have started to consider whether some semantics in the world of Web services can help us find the requested services. The first thing to do is to take another look at the Web service standards and see if there is already any semantics information present for us to use.

Altogether there are three main components. Let us examine them one by one to see if any of them can help. The first thing we notice is that SOAP is mainly for the lower-level data exchanging and communicating; it is the work after the discovery, so it will not help us much in the automatic discovery part. There are then two major pieces left: WSDL and UDDI.

WSDL is the description of a Web service, and it could be the perfect place to dig for more information. However, there is no semantics inside a WSDL document. Recall the Web service example about the megapixel value of a given camera. Examining its WSDL document (List 11.2), you can see that the only semantic information in it is that this service takes an `xsd:string` as input and returns an

`xsd:double` as output. This will not help at all, because many other completely different Web services might use exactly the same parameter sets.

A closer look at UDDI leads us to the same conclusion. For our example `getMegaPixel` Web service, UDDI offers a `businessEntity` instance containing a `businessService` entry, which provides the `bindingTemplate` and, finally, contains a `tModelKey` pointing to the interface `tModel`. However, the `tModel` mainly provides a reference to the corresponding WSDL document we just examined. Therefore, UDDI does not have any built-in semantics, either.

Thus, the solution to the automatic discovery problem is to explicitly add semantics to either the WSDL document or the UDDI registry. This is indeed a possible direction to take, and the rest of this book will concentrate on this direction.

Now, after a long detour, we finally get back to our track. Yes, there is a name for this new breed of Web services: they are called Semantic Web services. To be more specific, they are Web services with explicit semantic annotation. The vision is to apply semantic descriptions to Web services to provide relevant criteria for their automated discovery.