14 A Search Engine for Semantic Web Services

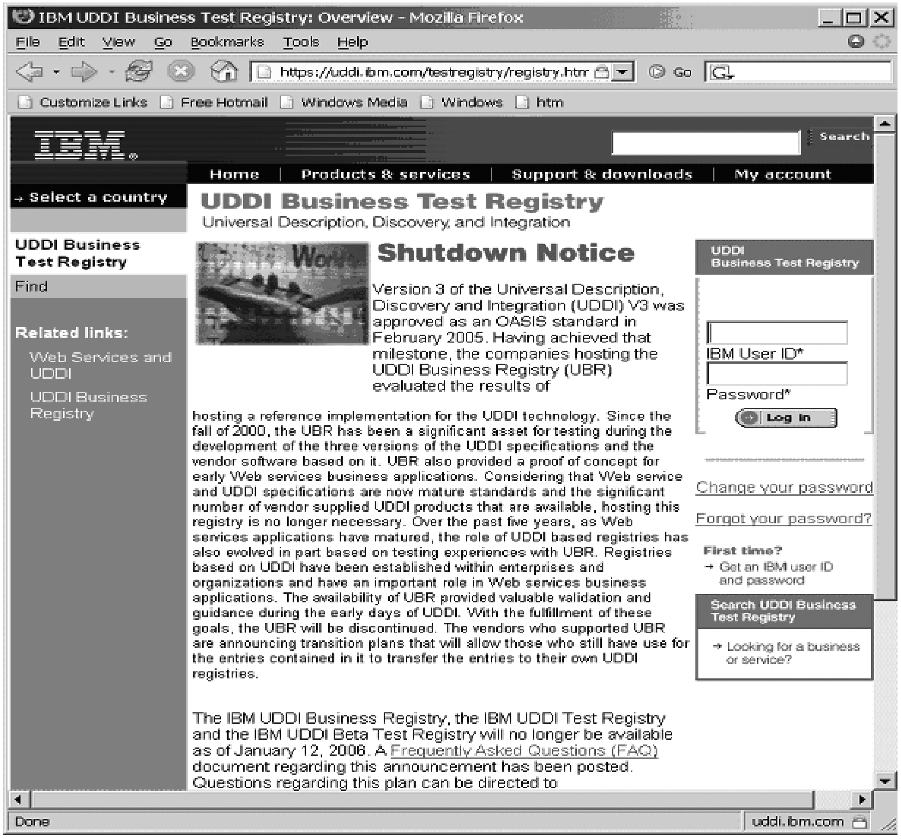
14.1 THE NEED FOR SUCH A SEARCH ENGINE

In the previous chapters, we have learned the following:

1. To make automatic service discovery possible, you need to add semantics to the Web service descriptions:
   - You can use WSDL-S (a lightweight solution) to accomplish this.
   - You can also use OWL-S upper ontologies (a full solution) to semantically describe a service.
2. All the semantically enhanced service descriptions have to be collected into some registry to facilitate the discovery:
   - Universal Description, Discovery, and Integration (UDDI) has been selected as the registry.
   - The details of mapping OWL-S profile document into the UDDI data structure have been discussed.
   - The resulting UDDI is called a semantically enhanced UDDI.
3. A matching algorithm has to be designed to search for the candidate services, based on the given request:
   - A matching algorithm specifies the semantic matching between a service request and a service description.
   - A piece of software that implements a given matching algorithm is called a matchmaking engine.

By putting all these pieces together, we can get a solution to the goal of automatic discovery of Web services. The backbone of this solution is clearly the UDDI registry.

But recently, public UDDI registries have been shut down by major players such as Microsoft, IBM, and SAP. Figure 14.1 shows the shutdown notice of IBM's test UDDI registry. This may or may not be a problem for us. Private UDDI registries continue to exist and are used by different organizations; semantically enhanced UDDI registries can still be built within the organization and used for automatic discovery of requested services. However, if the goal is to automatically search for a service over the Internet (certainly outside the domain of the organization), then it is just not possible; we would have to find another public registry to hold all the Semantic Web service descriptions.



UDDI Business Test Registry

Universal Description, Discovery, and Integration

UDDI Business Test Registry

Shutdown Notice

Version 3 of the Universal Description, Discovery and Integration (UDDI) V3 was approved as an OASIS standard in February 2005. Having achieved that milestone, the companies hosting the UDDI Business Registry (UBR) evaluated the results of hosting a reference implementation for the UDDI technology. Since the fall of 2000, the UBR has been a significant asset for testing during the development of the three versions of the UDDI specifications and the vendor software based on it. UBR also provided a proof of concept for early Web services business applications. Considering that Web service and UDDI specifications are now mature standards and the significant number of vendor supplied UDDI products that are available, hosting this registry is no longer necessary. Over the past five years, as Web services applications have matured, the role of UDDI based registries has also evolved in part based on testing experiences with UBR. Registries based on UDDI have been established within enterprises and organizations and have an important role in Web services business applications. The availability of UBR provided valuable validation and guidance during the early days of UDDI. With the fulfillment of these goals, the UBR will be discontinued. The vendors who supported UBR are announcing transition plans that will allow those who still have use for the entries contained in it to transfer the entries to their own UDDI registries.

The IBM UDDI Business Registry, the IBM UDDI Test Registry and the IBM UDDI Beta Test Registry will no longer be available as of January 12, 2006. A Frequently Asked Questions (FAQ) document regarding this announcement has been posted. Questions regarding this plan can be directed to

Related links:

Web Services and UDDI

UDDI Business Registry

UDDI Business Test Registry

IBM User ID*

Password*

Change your password

Forgot your password?

First time?

→ Get an IBM user ID and password

Search UDDI Business Test Registry

→ Looking for a business or service?

FIGURE 14.1 Shutdown notice of IBM's test UDDI.

This could dramatically change the whole work flow. More precisely, if the public UDDI registries were not shut down, a service provider could continue using some UDDI tools to map the semantic description of a given web service into UDDI entries. Also, a service requester could have used UDDI APIs or some other embedded matchmaking engine to automatically search for a requested service. In fact, some research has already been done along these lines [53].

Now, with the shutdown of UDDI public registries, the foregoing workflow is not feasible anymore. It is, however, not wise to follow the path of UDDI by creating yet another centralized repository and asking the potential service providers to publish their semantic service descriptions into this newly created registry.

A possible solution, however, is to take the burden away from both the service publishers and service requesters by making the whole publish–search cycle more transparent to all these users. More specifically, this can be done as follows:

- Service providers could simply publish their services using OWL-S or WSDL-S (or any other markup language) on their own hosting Web sites,

without worrying about using APIs or mapping tools to publish the services into some centralized repository.
- A Web crawler could collect all these semantic descriptions from their individual publishing sites and save them into a centralized repository, without the providers' knowledge; only this crawler would have knowledge about the specific structure of the registry.
- Service requesters could form and submit their service requests to the central registry; again, they should ensure the service requests are expressed using OWL-S or WSDL-S (or any other markup language); no knowledge about the structure of the registry is needed.
- The centralized registry assumes the responsibility of receiving the service requests, invoking some matchmaking engine, and returning a candidate set to the service requesters.

Such a solution will make it possible for a service requester to look for a desired service without even knowing whether it exists; the only requirement is to express the request in a semantic markup language such as OWL-S. Also, this solution will make it possible for a service provider to publish service descriptions directly on his or her Web server without even knowing how the service descriptions are collected and discovered; again, the service descriptions should be constructed using a markup language such as OWL-S.

This solution certainly does not need any public UDDI as its backbone support; it works almost like a search engine, except that it is a specialized search engine: a search engine for the automatic discovery of Semantic Web services.

In this chapter, we are going to design this search engine in detail, and we are also going to do some coding to implement it as a prototyping system to show how the added semantics can help a service requester find the required services automatically. We have reached the last part of this book, and we hope that the design of such a search engine and all the related coding exercises will help you to put together all that you have learned so far and will give you a more concrete and detailed understanding of the Semantic Web and Semantic Web services.

In the next section, we will discuss the design of the search engine; the implementation details are discussed in later sections in this chapter.

14.2 DESIGN OF THE SEARCH ENGINE

14.2.1 Architecture of the Search Engine

The design of the search engine is shown in Figure 14.2. The functionality of each component and the interaction among the components will be discussed now.

14.2.2 Individual Components

The first component for discussion is the Web crawler. The goal of the crawler is to collect the Semantic Web service descriptions; this is done by visiting the Web, finding the hosting Web sites, and parsing the service descriptions. These descriptions

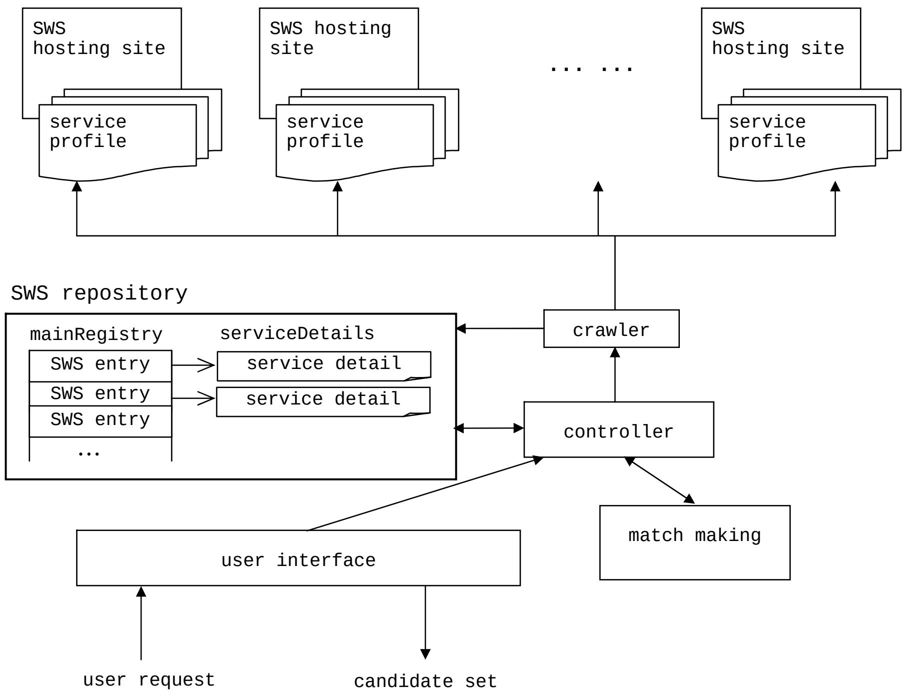


FIGURE 14.2 Structure of a search engine for Semantic Web Services.

can be written in OWL-S or WSDL-S, or in any other markup language. Clearly, if the service publisher does not offer a Semantic Web service description, the published Web service will not be collected into the Semantic Web service repository.

This crawler is an example of a *focused* crawler: it does not just collect everything it encounters; instead, it collects only the Semantic Web service descriptions. (We will see how the crawler decides whether a given page is such a description or not in the implementation section.) An obvious problem that needs to be considered is the "sparseness" of the service descriptions. In other words, there are not many Semantic Web service descriptions on the Internet for collection, and most pages the crawler visits would have nothing to do with Semantic Web services. Therefore, precious time and system resources could be wasted.

A possible solution to this problem is to come up with a better set of seed URLs. For instance, we can use Swoogle to search for all the documents that use the OWL-S upper ontology; the results returned by Swoogle are then used as an initial set of seed URLs. You can also design other heuristics to handle this problem.

The next component is the Semantic Web service repository. Its main purpose is to store the descriptions of Semantic Web services. It is similar to the index database in a traditional search engine. The structure of the repository is designed to provide enough information for the discovery process. Our current design of the repository contains two main tables. The first table is called `mainRegistry`, and its structure is shown in Table 14.1.

TABLE 14.1
Structure of the `mainRegistry` Table

Field Name	Meaning
`serviceID`	Key to identifying this service; the URL of the Semantic Web service description file is used as the value of this field
`ontologyURL`	Each Semantic Web service description has to be constructed based on some ontology; this is the URL of the ontology document
`serviceName`	Name of this service
`contactInfo`	Contact information of this service; it could be an e-mail address or phone number, fax number, etc.
`WSDLURL`	URL address of the WSDL document for this Web service

TABLE 14.2
Structure of the `serviceDetail` Table

Field Name	Meaning
`serviceID`	Foreign key linking back to the `mainRegistry` table
`parameterType`	This value shows whether this parameter is an input or output parameter
`parameterClass`	The ontology concept of this given parameter

All these fields are quite obvious, and the only one that requires some explanation is the `WSDLURL` field. Note that even when a service provider semantically marks up the service description, there would still exist the WSDL document for the given service. It is a good idea to include a link to this WSDL document in the repository for reference. By the same token, you can include the URLs pointing to the process and grounding documents in case the description file is written using the OWL-S upper ontology. The point is, change the design as you wish and experiment with it to learn more about Semantic Web services.

As you can see, the `mainRegistry` table only saves the basic information about a given service; the details about this service are stored in the `serviceDetail` table. Its structure is shown in Table 14.2.

Let us use our `getMegaPixel` service to illustrate how the information in these two tables is applied. In the `mainRegistry` table, our service will have a record that looks like this:

```
serviceID: http://localhost/chap11/GetMegaPixelProfile.owl
ontologyURL: http://www.yuchen.net/photography/Camera.owl
serviceName: getMegaPixel
contactInfo: lyu2@tinman.cs.gsu.edu
WSDLURL: null
```

The service details are saved in the `serviceDetail` table using two records:

```
serviceID: http://localhost/chap11/GetMegaPixelProfile.owl
parameterType: INPUT-0
```

```
parameterClass: model
serviceID: http://localhost/chap11/GetMegaPixelProfile.owl
parameterType: OUTPUT-0
parameterClass: pixel
```

Clearly, by combining the related records from these two tables, you get a complete description of the service. Note that the crawler only collects the `input` and `output` information; i.e., there is no `precondition` and `effect` information. Therefore, the matching algorithm in this search engine is based on `input` and `output` information only. Again, you can always add more information and experiment.

The next component is the matchmaking engine. It is the component that connects the domain ontologies, the repository, and the service requests. Using its built-in algorithm, it is responsible for recommending potential services based on a given request. More specifically, upon receiving a service request, the engine will read the detailed information of the request, access the tables in the repository, apply the algorithm to analyze each available service in the repository, and decide its qualification. We will present a matching algorithm in the next section.

The user interface is the point of contact between the user and the search engine. The user submits the search request by specifying a URL indicating the location of the service request description document. This request document semantically describes the requested service. For example, OWL-S can be used to construct such a request document, and we will see such an example later.

The final piece is the controller component. First, it creates and updates the repository through management of the Web crawler. Second, it accepts a service request from the user, invokes the matchmaking engine to access the repository, and returns a candidate set of potential Semantic Web services that might satisfy the request.

Putting all these components together, it is clear that the performance of the search engine is greatly influenced by the quality of the matching algorithm. We will propose a simple matching algorithm in the next section and use examples to make everything clear to you.

14.2.3 A Matchmaking Algorithm

In this section, we will discuss a matchmaking algorithm in detail, which we will implement in the next section.

A matchmaking algorithm is normally proposed in the following context:

- A domain-specific ontology is created to express the semantic capabilities of services, and these services are described with their inputs and outputs.
- For service discovery, clients and providers should use the same ontology to describe requests and services.

The first assumption is not new to us; we know that you have to markup your service description using some ontology. The second assumption is important to remember: if clients and providers use different ontologies, they are then talking in different languages; there is no shared understanding of the context, and it is simply not possible for the matchmaking algorithm to work.

Before we get into the details of the matching algorithm, let us first introduce some notations to simplify the presentation of the proposed algorithm. These notations are summarized in Table 14.3. Further, let C be any class or property, C = C(e), such that $\forall e \in (I_R \cup I_P \cup O_R \cup O_P)$, $C \in \Omega$. In other words, every input and output concept in the description files can be mapped to a single class or concept, namely, C, defined in the domain-specific ontology. Given all these definitions, we further need to define some useful mapping functions for the input set; these functions are listed in Table 14.4.

Clearly, $f_{input\text{-}exact}$ represents an "exact" matching case. In other words, this exact matching can be expressed as follows:

- The number of input parameters required by the service is the same as that offered by the service requester.
- For each input class or concept in the requester's input parameter set, there is an equivalent input class or concept in the provider's input parameter set.

TABLE 14.3
Notations for Our Simple Matchmaking Algorithm

Variable Name	Meaning
I_R	The set of input concepts or classes provided by the service requester
O_R	The set of output concepts or classes provided by the service requester
I_P	The set of input concepts or classes in the service description file offered by the service provider
O_P	The set of output concepts or classes in the service description file offered by the service provider
Ω	The set of all the concepts or classes that are defined in the domain-specific ontology. Therefore, it is true that $I_R \subseteq \Omega$, $O_R \subseteq \Omega$, $I_P \subseteq \Omega$, and $O_P \subseteq \Omega$

TABLE 14.4
Input Mapping Functions Defined for Our Simple Matchmaking Algorithm

Function Name	Meaning
$f_{input\text{-}exact}$	A 1-1 mapping from $I_R \rightarrow I_P$: $\forall e_{iR} \in I_R$, $\exists e_{iP} \in I_P$, such that $C(e_{iR}) \equiv C(e_{iP})$. "$\equiv$" means the left-hand-side (LHS) concept is equivalent to the right-hand-side (RHS) concept; clearly, this implies that $\lvert I_R \rvert = \lvert I_P \rvert$, i.e., the number of input concepts given by the service requester is equal to that required by the Web service provided by the service publisher
$f_{input\text{-}L1}$	A 1-1 mapping from $I_R \rightarrow I_P$: $\forall e_{iR} \in I_R$, $\exists e_{iP} \in I_P$, such that $C(e_{iR}) < C(e_{iP})$. "<" means the LHS concept is a subconcept or subclass of the RHS concept; clearly, this implies that $\lvert I_R \rvert = \lvert I_P \rvert$
$f_{input\text{-}L2}$	A 1-1 mapping from $I_R \rightarrow I_P$: $\forall e_{iR} \in I_R$, $\exists e_{iP} \in I_P$, such that $C(e_{iR}) > C(e_{iP})$; ">" means the LHS concept is a superconcept or superclass of the RHS concept; clearly, this implies that $\lvert I_R \rvert = \lvert I_P \rvert$

$f_{input\text{-}L1}$ can be read and understood in a similar way. It is, however, not as desirable as $f_{input\text{-}exact}$, and we thus call it a `level-1` matching. More precisely, there exists some `input` parameter in the requester's parameter set that is a subconcept or subclass of the required class or concept. For instance, a service could be expecting `Digital` as the `input` parameter, but the service requester offers an `SLR` class, a subclass of `Digital`, as `input`. This is clearly not an exact match, but it is acceptable. To see this, look at the class concepts in the object-oriented design world: a subclass is simply a more specialized version of the parent class; therefore, an instance of the subclass should already contain all the data members that might be needed by an instance of the parent class. In other words, you can make an instance of the parent class from an instance of the subclass; so it is perfectly alright to accept an instance of the subclass where an instance of the parent class is needed.

Now it is easier to understand $f_{input\text{-}L2}$, the `level-2` matching. It is just the opposite of `level-1` matching: the required input is a more specialized version than what the service requester can provide. For instance, a service could be expecting `SLR` as the `input` parameter, but the service requester offers a `Digital` class, a parent class of `SLR`, as `input`. This might still be a candidate because the service requester might have already provided enough information for the service to run. On the other hand, it is also possible that the service needs some specific information that an instance of `Digital` cannot provide.

If none of the foregoing three mapping functions hold when comparing the `input` sets, we can safely conclude that there is no match at all.

Similarly, three mapping functions can be analogously defined for the `output` concept set, as shown in Table 14.5.

Note the subtle differences between the `output` and `input` mapping functions. As the `input` mapping functions have been explained earlier, there is no need to further explain the `output` ones; they must be quite clear by now. With all these notations defined, the key part of the matching algorithm can now be described, as shown in List 14.1.

TABLE 14.5
`Output` Mapping Functions Defined for Our Simple Matchmaking Algorithm

Function Name	Meaning
$f_{output\text{-}exact}$	A 1-1 mapping from $O_R \rightarrow O_P$: $\forall e_{oR} \in O_R$, $\exists e_{oP} \in O_P$, such that $C(e_{oR}) \equiv C(e_{oP})$. "≡" means the left-hand-side (LHS) concept is equivalent to the right-hand-side (RHS) concept clearly, this implies that $\lvert O_R \rvert = \lvert O_P \rvert$, i.e., the number of output concepts given by the service requester is equal to the number of `input` concepts required by the Web service provided by the service publisher
$f_{output\text{-}L1}$	A 1-1 mapping from $O_R \rightarrow O_P$: $\forall e_{oR} \in O_R$, $\exists e_{oP} \in O_P$, such that $C(e_{oR}) > C(e_{oP})$. ">" means the LHS concept is a superconcept or superclass of the RHS concept clearly, this implies that $\lvert O_R \rvert = \lvert O_P \rvert$
$f_{output\text{-}L2}$	A 1-1 mapping from $O_R \rightarrow O_P$: $\forall e_{oR} \in O_R$, $\exists e_{oP} \in O_P$, such that $C(e_{oR}) < C(e_{oP})$; "<" means the LHS concept is a subconcept or subclass of the RHS concept clearly, this implies that $\lvert O_R \rvert = \lvert O_P \rvert$

LIST 14.1
A Simple Matchmaking Algorithm

```
Input: 1. Web service request description (.owl)
       2. current service description from the repository
Output: a string value from the set {"exact","level-1",
        "level-2","failed"}
Method:
build $I_R$,$O_R$ using Web service request description file;
build $I_P$,$O_P$ using the current SWS repository;
if $|I_R| \neq |I_P|$ or $|O_R| \neq |O_P|$ return "failed";
else if ( $\exists f_{input-exact}$ and $\exists f_{output-exact}$ ) return "exact";
else if ( $\exists f_{input-exact}$ and $\exists f_{output-L1}$ ) return "level-1";
else if ( $\exists f_{output-exact}$ and $\exists f_{input-L1}$ ) return "level-1";
else if ( $\exists f_{input-L1}$ and $\exists f_{output-L1}$ )   return "level-1";
else if ( $\exists f_{input-exact}$ and $\exists f_{output-L2}$ ) return "level-2";
else if ( $\exists f_{output-exact}$ and $\exists f_{input-L2}$ ) return "level-2";
else if ( $\exists f_{input-L1}$ and $\exists f_{output-L2}$ )   return "level-2";
else if ( $\exists f_{input-L2}$ and $\exists f_{output-L1}$ )   return "level-2";
else if ( $\exists f_{input-L2}$ and $\exists f_{output-L2}$ )   return "level-2";
else return "failed";
```

Again, this algorithm is easy to understand; in the next section, you will see an implementation of this algorithm. The basic idea is to first check the number of `inputs` and `outputs`; if the number of `inputs` (or `outputs`) of the required service is different from the number of `inputs` (or `outputs`) of the current candidate, the match immediately fails. If the numbers match, the algorithm goes on to check the mapping functions to determine the degree of matching. For instance, if exact matching functions can be found for both the `inputs` and `outputs`, then the current candidate service exactly matches the requirement. But if exact matching functions can be found for `inputs` (or `outputs`), but only `level-1` matching functions can be found for `outputs` (or `inputs`), then the matchmaking engine will declare a `level-1` matching, and so on.

Note that this simple matching algorithm considers mainly the matching between `inputs` and `outputs`. Other factors can also be taken into account. For example, the service category can also be considered a matching criterion, and it is not hard to extend this to the aforementioned algorithm. Also, you can take preconditions and effects of the Web service into consideration. We will not cover all these possibilities in this chapter; this algorithm serves as a starting point for your journey of creativity.

As we have discussed earlier, `level-1` and `level-2` matching results were proposed to differentiate the parent–child relationship of the concept. In the literature, this relationship is sometimes called *subsumption relationship* [54]. It is also obvious that the direction of this subsumption relation is important: in the case where input A in the request profile subsumes input B in the candidate service profile, the advertised service may require some specific details for proper execution that input A cannot provide. In a situation like this, our matching algorithm still includes this service as a potential candidate and lets the service requester decide whether the service is usable or not. This degree of matching is called a `level-2` matching; therefore, a `level-2` matching may not be as appropriate as a `level-1` matching.

Up to this point, we have discussed all the major components in the search engine; it is now time to do some implementation work.

14.3 IMPLEMENTATION DETAILS

14.3.1 Housekeeping Work

Before the real implementation work, preparation is necessary, such as setting up a seed URL for the Web crawler; we also need some utility classes. We will discuss these two parts in this section.

14.3.1.1 A Seed URL for the Web Crawler

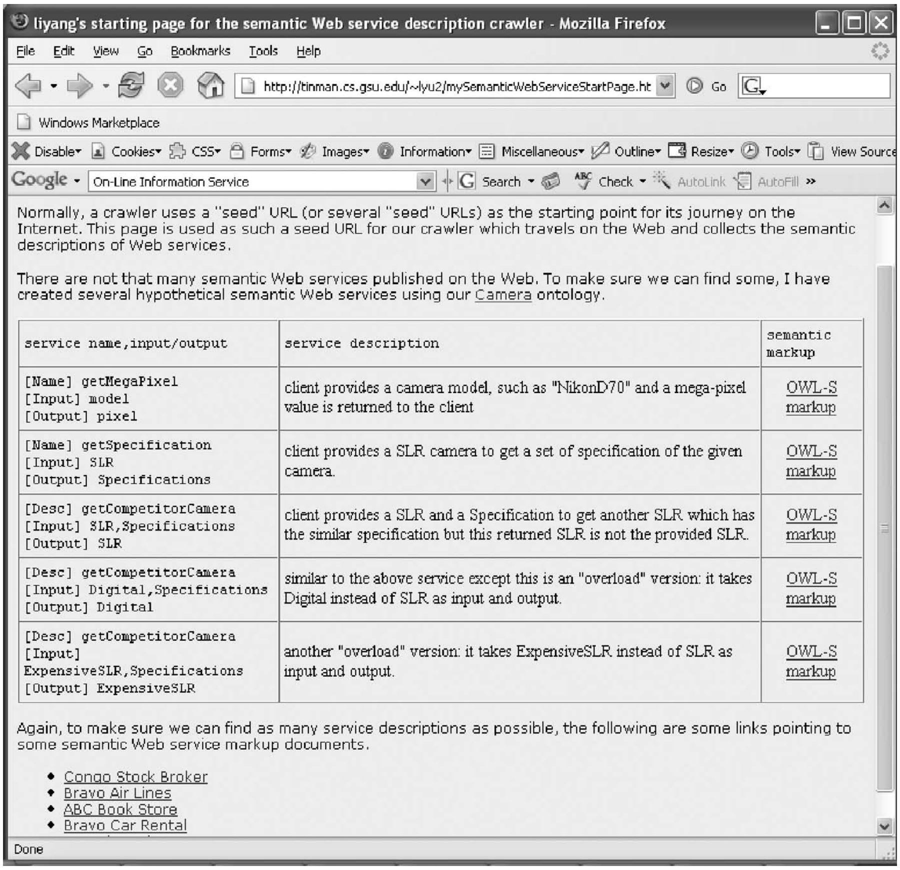
I assume that you will set up your search engine using your PC at home; therefore, you will not be able to allocate much memory or CPU time for the crawler. Given this limitation, how do we proceed (there are not many Semantic Web service descriptions on the Internet)? It is quite possible for our crawler to search the Internet for quite some time without finding any descriptions.

We will discuss other solutions later, but for now, let us create a Web page and use it as the seed URL for the crawler. On this page, we will have links pointing to several hypothetical Semantic Web service descriptions (we will create these documents later) and also to some other real Semantic Web service descriptions. The whole purpose of this starting page is to ensure that our crawler will be able to find some descriptions and collect them into our repository, given the limited resources. Figure 14.3 shows the page that I made for this search engine.

On our seed page, we have added links to four hypothetical Semantic Web service description documents. These documents are written in OWL-S, and they are also published on the Web; so the crawler will be able to follow the links and collect these descriptions.

You have seen the OWL-S description of the `getMegaPixel` service already. As another example, List 14.2 is the OWL-S description of the `getCompetitor-Camera` service. Note that only the most relevant parts of the description document are shown in List 14.2, including the basic service information and the IOPE section. For the other sections, you can see the `getMegaPixel` profile document in List 12.8.

The other description documents are similar, and we will not list them one by one. You can simply click the links in Figure 14.3 to read them. Among these links, there are also some real OWL-S service description documents; for example, the Bravo Air Lines is one such link. Again, these links are added to ensure that our crawler is able to find some descriptions.


liyang's starting page for the semantic Web service description crawler - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://tinman.cs.gsu.edu/~lyu2/mySemanticWebServiceStartPage.ht
Go
Windows Marketplace
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Google On-Line Information Service Search Check AutoLink AutoFill
Done

Normally, a crawler uses a "seed" URL (or several "seed" URLs) as the starting point for its journey on the Internet. This page is used as such a seed URL for our crawler which travels on the Web and collects the semantic descriptions of Web services.

There are not that many semantic Web services published on the Web. To make sure we can find some, I have created several hypothetical semantic Web services using our Camera ontology.

service name,input/output	service description	semantic markup
[Name] getMegaPixel [Input] model [Output] pixel	client provides a camera model, such as "NikonD70" and a mega-pixel value is returned to the client	OWL-S markup
[Name] getSpecification [Input] SLR [Output] Specifications	client provides a SLR camera to get a set of specification of the given camera.	OWL-S markup
[Desc] getCompetitorCamera [Input] SLR,Specifications [Output] SLR	client provides a SLR and a Specification to get another SLR which has the similar specification but this returned SLR is not the provided SLR.	OWL-S markup
[Desc] getCompetitorCamera [Input] Digital,Specifications [Output] Digital	similar to the above service except this is an "overload" version: it takes Digital instead of SLR as input and output.	OWL-S markup
[Desc] getCompetitorCamera [Input] ExpensiveSLR,Specifications [Output] ExpensiveSLR	another "overload" version: it takes ExpensiveSLR instead of SLR as input and output.	OWL-S markup

Again, to make sure we can find as many service descriptions as possible, the following are some links pointing to some semantic Web service markup documents.

- Congo Stock Broker
- Bravo Air Lines
- ABC Book Store
- Bravo Car Rental

FIGURE 14.3 The seed page for our Web crawler.

LIST 14.2
The Profile Document for the `getCompetitorCamera` Service

```
<profile:serviceName>getCompetitorCamera</profile:serviceName>
<profile:textDescription>
   find another camera which has the similar performance
</profile:textDescription>

<profile:contactInformation>
  <actor:Actor rdf:ID="getCompetitorCamera_provider">
     ... provider information ...
  </actor:Actor>
</profile:contactInformation>
```

```
<!-- IOPE --!>
<profile:hasInput>
   <process:Input rdf:ID="input_slr">
     <process:parameterType>
        http://www.yuchen.net/photography/Camera.owl#SLR
     </process:parameterType>
   </process:Input>
</profile:hasInput>
<profile:hasInput>
   <process:Input rdf:ID="input_spec">
     <process:parameterType>
        http://www.yuchen.net/photography/Camera.owl#Specifications
     </process:parameterType>
   </process:Input>
</profile:hasInput>
<profile:hasOutput>
   <process:Output rdf:ID="output_slr">
      <process:parameterType>
         http://www.yuchen.net/photography/Camera.owl#SLR
      </process:parameterType>
   </process:Output>
</profile:hasOutput>
```

14.3.1.2 Utility Classes

To construct our search engine, we will need some utility classes, for example, a class that is responsible for downloading each page from a given URL. This section will cover these utility classes.

The first is the `myHTTPManager` class. It is responsible for downloading a Web page from a given URL. The major part of this class is summarized in List 14.3. We will not discuss this utility class further. It is a standard Java program with many comments, and I assume you will study it if you have trouble understanding it.

LIST 14.3
The Definition of Class `myHTTPManager`

```
/*
 * myHTTPManager.java
 * Created on September 7, 2005, 7:53 PM
 */

import java.net.*;
import java.io.*;

public class myHTTPManager
{
   public final static int HTTP_PORT = 80;  // well-known WWW port
```

```
/** Creates a new instance of myHTTPManager */
public myHTTPManager()
{
}

public String downloadPage(URL pageURL)
{
   String line;                // variable for use within member func
   InputStream pStream = null; // variable for use within member func
   StringBuffer thePage = new StringBuffer(); // the downloaded page!

   try
   {
      pStream = getPageStream(pageURL); // downloading is done here!
      if (pStream == null) return "";
   }
   catch (Exception error)
   {
      System.out.println("get(host, file) failed!" + error);
      return "";
   }

   BufferedReader br =
   new BufferedReader(new InputStreamReader(pStream));
   try
   {
      while ((line = br.readLine())!= null) thePage.append(line);
      br.close();
   }
   catch (Exception error) { }

   // returned the downloaded page
   return thePage.toString();
}

/*
 * the real downloading happens here
 */
public InputStream getPageStream(URL url)
       throws IOException, UnknownHostException
{
   // necessary protection
   if ( url.getPort() == -1 )
      url =
      new URL(url.getProtocol(),url.getHost(),HTTP_PORT,
      url.getFile());

   // create a socket, connect it to specified port number on web host
   Socket socket = new Socket(url.getHost(),url.getPort());
```

```
      // throw a java.io.InterruptedIOException if the timeout expires
      socket.setSoTimeout(120000);

      // get a print stream from the output stream for this socket
      PrintStream out = new PrintStream(socket.getOutputStream());

      // construct HTTP GET request line
      out.print("GET " + url.getFile() + " HTTP/1.0\n");

      // tell the server the referenced URL from which the request URL
      // was obtained so that the obsolete or mistyped link can be
      // traced for maintenance
      out.print("Referer: " + url.getPath() + "\r\n");

      // tell the server our crawler's name so the server can
      // distinguish it from browsers
      out.print("User-Agent: myWebCrawler/1.0" + "\r\n");

      // provide an email so that server can contact us in case of
         problems
      out.print("From: lyu2@tinman.cs.gsu.edu" + "\r\n");
      out.print("Pragma: no-cache" + "\r\n");   // ignore the caches

      // provide host and port number of the relative URL being
         requested
      out.print("Host: " + url.getHost() + ":" + url.getPort() +
      "\r\n");

      // accept all media types and their subtypes
      out.print("Accept: */*" + "\r\n");

      // a blank line indicates the end of the header fields
      out.print("\r\n");

      // flush the stream
      out.flush();

      // get the message from server – this is the page!
      InputStream in = socket.getInputStream();

      // close this socket. cannot do this here, let it die itself
      // socket.close();

      // return the page
      return in;
   }
} // end of class myHTTPManager
```

Another utility class is needed for managing the threads. A crawler is often implemented as a multithread application: to improve the efficiency, there are normally several crawlers visiting the Web simultaneously. The easiest way to create new threads is to let the current crawler create another crawler: after parsing the pages, the current crawler normally collects several new pages (links), and to visit each of these new pages, the crawler creates a new crawler.

Clearly, this will quickly get out of control. Soon there will be too many crawlers wandering on the Web, and too many threads running means overheads, which may hurt efficiency. Therefore, there has to be a way to control the number of living threads.

The next service class, `myThreadController`, is created for this. The basic idea is again quite simple: each thread, before it goes live, will ask for a "ticket" from the thread controller; if a ticket is available, this thread will obtain one and start to run. It will return the ticket after it finishes the work. On the other hand, if there is no available ticket, this thread will wait (sleep) until a ticket is available. Clearly, we can control the number of living threads in our system by simply controlling the total available number of tickets. List 14.4 is the implementation of this service class. Again, this simple utility class does not need much explanation. With these two helper classes constructed, we can move on to the key components of the search engine.

LIST 14.4
The Definition of `myThreadController` Class

```
/*
 * myThreadController.java
 * Created on September 7, 2005, 8:39 AM
 */

public class myThreadController
{
   int tickets[];           // available tickets
   int currentThreadCount;  // number of current running threads
   int myThreadCount;       // max number of running threads

   public myThreadController(int myThreadCount)
   {
       this.myThreadCount = myThreadCount;
       currentThreadCount = 0;

       tickets = new int[myThreadCount+1];
       for(int i=0; i < myThreadCount; i++)
       {
          tickets[i] = -1;  // -1 means available
       }
   }
```

```
    /*
     * get a ticket – permission to run
     */
    public synchronized int getTicket()
    {
        while (currentThreadCount == myThreadCount)
        {
            try { wait(); }
            catch (InterruptedException leaveUsAlonePlease) {}
        }

        // once we reach here, there is for sure ticket(s) available
        int ticket = findFreeTicket();
        tickets[ticket] = ticket; // mark the ticket unavailable
        currentThreadCount++;     // increase the number of running
                                     thread
        return ticket;
    }

    /*
     * returning the ticket when it is done
     */
    public synchronized void returnTicket(int ticket)
           throws IllegalArgumentException
    {
        tickets[ticket] = -1;  // mark ticket as available again.
        currentThreadCount--;  // decrease the number of running threads
        notifyAll();           // wake up a thread needing a ticket
    }

    // Find any ticket which hasn't been issued yet.
    protected int getFreeTicket()
    {
        for(int i=0; i < myThreadCount; i++)
        {
            if (tickets[i] == -1) return i;
        }
        return -1;
    }
}
```

14.3.2 Implementation of the Semantic Service Description Crawler

Let us now discuss the implementation details of the crawler. The crawler code is shown in List 14.5. First, note that the constructor accepts a `String` parameter named `pageName`, which represents the seed URL for the crawler. To change this `String` into a URL, the constructor follows the best practice: i.e., it first creates a URI instance from the `String` and then changes the URI instance into a URL instance

LIST 14.5
The Definition of `myWebCrawler` Class

```
/*
 * myWebCrawler.java
 * Created on September 7, 2005, 8:36 AM
 */
import java.util.*;
import java.io.*;
import java.net.*;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class myWebCrawler extends Thread
{
   private final static int MAX_THREADS = 4;
   private final static int MAX_VISITED_PAGES = 5000;
   private static int numOfInstance = 0;
   private int instanceID;
   public int getNumOfInstance() { return numOfInstance; }
   static myThreadController myThreadManager
          = new myThreadController(MAX_THREADS);

   // this is to remember the pages already visited
   static Hashtable visitedPages = new Hashtable();
   static Vector interestedLinks = new Vector(); // owl documents
   URL pageToFetch; // other service variable

   // log file
   static final String InfoReport = ">>> ";
   static final String logName = "myCrawlerLog.txt";
   static PrintWriter myLog = null;

   // report the interesting sites(owl sites).
   static int lastReportedSize = 0;
   static boolean reported = false;

  /** Creates a new instance of myWebCrawler */
   public myWebCrawler(String pageName)
   {
      URI uri = null;
      pageToFetch = null;
      instanceID = numOfInstance;
      numOfInstance ++;

      try
      { uri = new URI(pageName); }
      catch (URISyntaxException e)
      { System.out.println("error in URI format:" + pageName); }
      try
      { if ( uri != null ) pageToFetch = uri.toURL(); }
      catch (IllegalArgumentException e)
      { System.out.println(pageName + ":invalid URL ...
                   will not starting thread for this one!"); }
      catch (MalformedURLException badURL)
      { System.out.println(pageName + ": invalid URL ...
                   will not starting thread for this one!"); }

      // label this thread with the page name
      setName("[thread-" + instanceID + ":" + pageName + "]");
      // start the thread at run()
      start();
   }

   public void run()
   {
      int ticket;       // thread can run after getting a ticket
      String webPage;   // an entire Web page cached in a String
      Vector pageLinks; // bag to accumulate found URLs in

      ticket = myThreadManager.getTicket();
      if ( this.visitedPages.size() > MAX_VISITED_PAGES )
      {
         System.out.println(InfoReport + "reach the page limits,
                            stopping a crawler thread...");
         if ( myLog != null )
         {
            myLog.println(InfoReport + "reach the page limits,
                          stopping a crawler thread...");
            myLog.flush();
         }
         myThreadManager.returnTicket(ticket);
         return;
      }

      webPage = downloadPage(pageToFetch);
      pageLinks = extractLinks(webPage);

      if ( myLog != null )
      {
         myLog.println();
         myLog.println(InfoReport + getName() + " has " +
                       pageLinks.size() + " links.");
         myLog.flush();
      }
      System.out.println(getName() + " has " + pageLinks.size() + "
                         links.");

      Enumeration allLinks = pageLinks.elements();
      while( allLinks.hasMoreElements() )
      {
         String page = (String)allLinks.nextElement();

         if ( ! alreadyVisited(page) )
         {
            markAsVisited(page);
            String threadName =
                   getName().substring(0,getName().indexOf(':'));
            System.out.println(threadName + "] visiting-> "
            + page);

            if ( myLog != null )
            {
                myLog.println(threadName + "] visiting-> " + page);
                myLog.flush();
            }

            new myWebCrawler(page);
            System.out.println(InfoReport + "already visited: "
                               + visitedPages.size());
         }
         else
         {
            // System.out.println("Already visited: " + page);
         }
      }  // end of while loop

      myThreadManager.returnTicket(ticket);
      try // insert some random delay to give a chance to others
      { Thread.sleep( (int) (Math.random()*200) ); }
      catch (Exception e) {}

      if ( reported==false||lastReportedSize<interestedLinks
         .size() )
      {
         reported = true;
         lastReportedSize = interestedLinks.size();

         System.out.println("----------------------------------");
         Enumeration results = interestedLinks.elements();
         while( results.hasMoreElements() )
         {
            String page = (String)results.nextElement();
            System.out.println(page);
         }
         System.out.println("----------------------------------");
      }

   }
```

```
// prepare the log file
public static void createLog(String startURL)
{
   try
   {
      myLog = new PrintWriter(new FileOutputStream(logName),
              true);
      myLog.println("Semantic Web Services Indexation and
                     Discovery: OWL-S crawler log file");

      // report time
      Date myDate = new Date();
      myLog.println("DATE: " + myDate.toString());
      myLog.println();

      // important system parameters
      myLog.println(InfoReport + "starting URL:" + startURL);
      myLog.println(InfoReport + "max number of pages to
      visit:  "
                    + MAX_VISITED_PAGES);
      myLog.println(InfoReport + "max number of existing
      threads:"
                    + MAX_THREADS);
      myLog.println();
      myLog.flush();
   }
   catch( FileNotFoundException fnfex )
   {
      System.out.println("cannot create the log file!");
   }
}

// Given a valid URL, download the page as a big String
protected String downloadPage(URL page)
{
   myHTTPManager http = new myHTTPManager();
   return http.downloadPage(page);
}

protected Vector extractLinks(String page)
{
   String SubDomain = "(?i:[a-z0-9]|[a-z0-9][-a-z0-9]*
                      [a-z0-9])";
   String TopDomain = "(?x-i:com\\b        \n" +
                      "      |edu\\b        \n" +
                      "      |biz\\b        \n" +
                      "      |in(?:t|fo)\\b \n" +
                      "      |mil\\b        \n" +
                      "      |net\\b        \n" +
                      "     |org\\b        \n" +
                      "     |[a-z][a-z]\\b \n" + // country code
                      ")                   \n";
   String Hostname = "(?:" + SubDomain + "\\.)+" + TopDomain;
   String NOT_IN   = ";\"'<>()\\[\\]\\{\\}\\s\\x7F-\\xEF";
   String NOT_END  = "!.,?";
   String ANYWHERE = "[^" + NOT_IN + NOT_END + "]";
   String EMBEDDED = "[" + NOT_END + "]";
   String UrlPath  = "/" + ANYWHERE + "*(" + EMBEDDED + "+" +
                     ANYWHERE + "+)*";
   String Url =
   "(?x:                                                   \n" +
   "  \\b                                                  \n" +
   "  ## match the hostname part                           \n" +
   "  (                                                    \n" +
   "    (?: ftp | http s? ): // [-\\w]+(\\.\\w[-\\w]*)+  \n" +
   "   |                                                   \n" +
   "    " + Hostname + "                                   \n" +
   "  )                                                    \n" +
   "  # allow optional port                                \n" +
   "  (?: \\d+ )?                                          \n" +
   "                                                       \n" +
   "  # rest of url is optional, and begins with /         \n" +
   " (?: " + UrlPath + ")?                                 \n" +
   ")";

  // convert string we just built up into a real regex object
  Pattern UrlRegex = Pattern.compile(Url);
  // ready to apply to raw text to find urls
  Matcher m = UrlRegex.matcher(page);
  // find everything
  Vector bagOfLinks = new Vector();
  while ( m.find() )
  {
     String theLink = m.group();
     if ( theLink == null ) continue; // protection
     // skip some of these links
     if ( theLink.endsWith(".gif") || theLink.endsWith(".jpg")
          || theLink.endsWith(".pdf") ) continue;
     // save the links we want!!
     if ( theLink.endsWith(".owl") )
     {
        if ( interestedLinks.indexOf(theLink) == -1 )
           interestedLinks.addElement(theLink);
        myRegistryBuilder.getInstance().buildRegistry(theLink);
     }
     // add this to the links yet to visit
     bagOfLinks.addElement(theLink);
  }
  return bagOfLinks;
```

```
161:   }

162:   protected boolean alreadyVisited(String pageAddr)
163:   {
164:      return visitedPages.containsKey(pageAddr);
165:   }

166:   protected void markAsVisited(String pageAddr)
167:   {
168:      visitedPages.put(pageAddr,pageAddr);
169:   }
170: }
```

named `pageToFetch` (lines 24 to 37). For each page yet to be visited, there will be a crawler thread created; therefore, a good name for this crawler thread is the URL of the page (line 38). It is not necessary to give a name to each thread; we do this just for reporting purposes. After these are done, the constructor calls the `start()` method to schedule this crawler thread for CPU time (line 39).

When it is time to run this crawler thread, its `run()` method is called (line 41). The first thing the crawler does is to try to get a ticket and start itself (line 46). Once it gets the ticket, it checks how many pages have been visited at that moment. If the page number has reached the page limit (we use this to limit the crawling time), the crawler will simply give up and return (lines 47 to 56).

If the number has not yet reached the limit, the crawler continues by downloading the page and extracting all the links in this page (lines 58 and 59). Note that lines 60 to 65 write the action of the crawler into a log file, so you know what exactly is happening in the system. It is always a good idea to have a log for this type of system.

Now, the crawler gets the first link; if the page pointed to by this link has already been visited, the crawler moves on to the next link. Otherwise, the crawler marks this page as having been visited already, creates another crawler thread, and passes this page to the newly created thread (lines 67 to 88). The crawler then moves on to the next link and repeats the same process until there are no more links to visit. This crawler thread has now finished its task; it returns the ticket to the thread manager, and goes to sleep a little bit to give other waiting threads a chance to start (lines 89 to 92).

At this point, it should be clear that we have many crawlers in the system; if there are ten unvisited links on the current page, the current crawler thread will create another ten crawler threads, one for each unvisited link. These ten new crawlers will perform the same steps, and other new crawlers will be created by them. Although there are many crawler threads in the system, most of them would be in the waiting state because only four crawlers are allowed to work simultaneously (line 8 specifies this limit).

Note that there are some reporting functionalities coded into the crawler; for instance, lines 93 to 104 report all the URLs that represent Semantic Web service descriptions at a particular moment. This information represents the "harvest" of our system, so it is very important to keep an eye on this figure. Also, a system log is created (lines 107 to 126) and used frequently in the system. We will not discuss these reporting schemas in more detail; you will understand them easily by reading the code or running the crawler.

To summarize, there are two main tasks in each crawler thread: downloading the page and getting all the links on it. Downloading a given page (lines 127 to 131) is done by using the utility class `myHTTPManager`; therefore, there is nothing new here. However, the real question is, when do we identify if a given page is a Semantic Web service description document or not? The answer is that it is done in the process of extracting the links.

Lines 132 to 161 implement this process. In fact, extracting the links from a page stream is not an easy task; you need to use regular expressions to do it. If you are not familiar with regular expressions, this might be the right time to get started. A good book on regular expressions is given in the reference list [55] at the end of the book.

Now, after getting all the links, we study them one by one before returning them to the crawler. If a given link points to a page that is of gif, jpg, or pdf format, we do not even return them to the crawler. The key line is line 155; if a given link points to a page that is an OWL document, we pass it to `myRegistryBuilder` class (discussed later) to see whether it is indeed a Semantic Web service description, and if it is, we get everything we need from it (again, details will be given later).

Now, we have seen all the main pieces of the crawler class. It may seem confusing at first glance. Well, the best course of action is to load the code into your PC and debug it to really understand it. Once you understand it, you will be able to change it and make it more efficient.

To run the crawler code, you need a driver. In my code, the search-engine controller component is the driver. However, in order not to make this chapter too long, I am not going to discuss the controller code; still, you can use the driver shown in List 14.6 to start the crawler code. This list gives a straightforward testing driver that does not require much explanation. Note that it makes use of `myRegistryBuilder` class several times. The real handling of the OWL files also happens in this class. Let us take a closer look at it in the next section.

LIST 14.6
Driver Class to Start the Crawler

```
/*
 * Main.java
 * Created on September 6, 2005, 7:47 PM
 */
public class Main {

    /** Creates a new instance of Main */
    public Main() { }

    /** @param args the command line arguments */
    public static void main(String[] args)
    {
      // specify the starting URL
      String startURL =
      "http://tinman.cs.gsu.edu/~lyu2/mySemanticWebServiceStartPage
      .html";

      // open up log file
      myWebCrawler.createLog(startURL);

      // connected to the database
      if ( myRegistryBuilder.buildMSAccessConnection()==false ) return;

      // start to crawl the web!!
      new myWebCrawler(startURL);

      // when there is only one thread again, we are done crawling
      int currentThreadCount = 0;
      do
      {
         try {  Thread.sleep(1000); }
         catch (Exception e) {}

         currentThreadCount = Thread.activeCount();
         if ( currentThreadCount == 1 ) break;
         else
           System.out.println(">>>>> current total thread number: " +
                              currentThreadCount );
      } while(true);

      // clean up
      System.out.println("");
      System.out.println("\n all done. ");
      System.out.println(">>>>>> shutdown database connection... ");
      myRegistryBuilder.shutdownDBConnection();
    }
}
```

14.3.3 Implementation of the Semantic Service Description Repository

The purpose of the repository is to hold the semantic descriptions, and you have seen the structures of the two tables in the repository. Class `myRegistryBuilder` implements this repository, and it is given in List 14.7. `myRegistryBuilder` class includes three key member functions: `buildMSAccessConnection()`, `shutdownD-BConnection()`, and `buildRegistry()`. To make our implementation of this search engine easier, Microsoft's Access database was used. As you can see, `buildMSAc-cessConnection()` is the place where we create the connection to the Access database; it is all standard Java programming and much explanation is not needed.

LIST 14.7
The Definition of `myRegistryBuilder` Class

```
/*
 * myRegistryBuilder.java
 * Created on September 15, 2005, 8:07 PM
 */
import java.util.*;
import java.sql.*;

// Jena interface
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.rdf.model.*;

public class myRegistryBuilder
{
   // related to the ontology
   static private final String W3C_UPPER_ONT_PROFILE =
          "http://www.daml.org/services/owl-s/1.0/Profile.owl#";
   static private final String W3C_UPPER_ONT_PROCESS =
          "http://www.daml.org/services/owl-s/1.0/Process.owl#";

   // database connection, used to build the registry
   static private Connection dbConnection = null;

   // singleton design pattern to make sure only one builder exists
   static private myRegistryBuilder theInstance = null;
   static public myRegistryBuilder getInstance()
   {
      if ( theInstance == null )
         theInstance = new myRegistryBuilder();
      return theInstance;
   }

   // singleton design pattern, so this has to be private
   private myRegistryBuilder() { }

   // build database connection
   static public boolean buildMSAccessConnection()
   {
      try
      {
         String url = "jdbc:odbc:myWebServiceRegistry";
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         dbConnection = DriverManager.getConnection(url);
      }
      catch (ClassNotFoundException cex)
      {
         cnfex.printStackTrace();
```

```
          System.out.println("db connection failed:" + cex
          .toString());
          return false;
       }
       catch (SQLException qex)
       {
          sqlex.printStackTrace();
          System.out.println("db connection failed:" + qex
          .toString());
          return false;
       }
       return true;
    }

    static public void shutdownDBConnection()
    {
       try { dbConnection.close(); }
       catch (SQLException e)
       {
          System.out.println("db shutdown failed: " +
          e.getMessage());
          return;
       }
    }

    public void buildRegistry(String owlURL)
    {
       // use Jena to load the owl document
       Model model = ModelFactory.createDefaultModel();
       try { model.read(owlURL); }
       catch(Exception e)
       {
          System.out.print(owlURL);
          System.out.println(": has a problem accessing it, abort.");
          return;
       }

       // confirm this owl document is indeed a semantic description
       String rightPrefix = "profile";
       if ( rightPrefix.equalsIgnoreCase
            (model.getNsURIPrefix(W3C_UPPER_ONT_PROFILE)) == false )
       {
          System.out.println(owlURL + " is not a semantic web
          service
                             description file, ignored.");
           return;
       }
       else
       {
          Property inputs =
                model.getProperty(W3C_UPPER_ONT_PROFILE +
                "hasInput");
          Property outputs =
                model.getProperty(W3C_UPPER_ONT_PROFILE +
                "hasOutput");
          if ( inputs == null && outputs == null )
          {
             System.out.println(owlURL + " is not a semantic web
                              service description file, ignored.");
             return;
          }
       }

       // now let us find the input and outputs

       Vector inputName = new Vector();
       Vector outputName = new Vector();
       Vector inputClass = new Vector();
       Vector outputClass = new Vector();
       String serviceName = null;
       String contactInfo = null;

       // first scan: find all the input and output names
       StmtIterator iter = model.listStatements();
       while ( iter.hasNext() )
       {
          com.hp.hpl.jena.rdf.model.Statement stmt
                                             = iter.nextStatement();
          String subjectString = stmt.getSubject().toString();
          String predicateString = stmt.getPredicate().toString();
          String objectString = stmt.getObject().toString();
          if ( objectString.equals(W3C_UPPER_ONT_PROCESS+"Input") )
          {
             String iName = null;
             iName =
             subjectString.substring(subjectString
             .lastIndexOf('#')+1);
             if ( iName != null )
             {
                if ( inputName.contains(iName) == false )
                   inputName.addElement(iName);
             }
          }
          else if
    (objectString.equals(W3C_UPPER_ONT_PROCESS+
     "UnConditionalOutput") )
          {
             String oName = null;
             oName =
             subjectString.substring(subjectString.lastIndexOf
             ('#')+1);
```

```
            if ( oName != null )
            {
               if ( outputName.contains(oName) == false )
                  outputName.addElement(oName);
            }
         }
         else if ( predicateString.endsWith("serviceName") )
            serviceName = stmt.getObject().toString();
         else if ( predicateString.endsWith("email") )
            contactInfo = stmt.getObject().toString();
      }

      // second scan: find all the classes/concepts for the input
      String domainOntURL = null;
      for ( int i = 0; i < inputName.size(); i ++ )
      {
         String inputName = inputName.elementAt(i).toString();
         iter = model.listStatements();
         while ( iter.hasNext() )
         {
            com.hp.hpl.jena.rdf.model.Statement stmt
                                           = iter.nextStatement();
            String sbjt = stmt.getSubject().toString();
            String prdt = stmt.getPredicate().toString();
            String objt = stmt.getObject().toString();
            if (
         sbjt.substring(sbjt.lastIndexOf("#")+1).equals
         (inputName) &&
         prdt.equals(W3C_UPPER_ONT_PROCESS+"parameterType") )
            {
               if ( domainOntURL == null )
                  domainOntURL =
                  objt.substring(0,objt.lastIndexOf('#'));
               inputClass.addElement
                          (objt.substring(objt.lastIndexOf('#')
                           +1));
               break;
            }
         }
      }

      // third scan: find all the classes/concepts for the output
      for ( int i = 0; i < outputName.size(); i ++ )
      {
         String outputName = outputName.elementAt(i).toString();
         iter = model.listStatements();
         while ( iter.hasNext() )
         {
            com.hp.hpl.jena.rdf.model.Statement stmt
                                           = iter.nextStatement();
            String sbjt = stmt.getSubject().toString();
            String prdt = stmt.getPredicate().toString();
            String objt = stmt.getObject().toString();
            if (
         sbjt.substring(sbjt.lastIndexOf("#")+1).equals
         (outputName) &&
         prdt.equals(W3C_UPPER_ONT_PROCESS+"parameterType") )
            {
               outputClass.addElement
                          (objt.substring(objt.lastIndexOf('#')
                           +1));
               break;
            }
         }
      }

      // error protection
      if ( inputClass.size() == 0 && outputClass.size() == 0 )
      return;

      // write to the registry: main registry
      String updateStmt = "";
      try
      {
         java.sql.Statement stmt = dbConnection.createStatement();
         updateStmt = "insert into mainRegistry (" +
         "serviceURL, ontologyURL, serviceName, contactEmail,
         WSDLURL"
         + ") values ('" + owlURL + "','" + domainOntURL + "','"
         + serviceName + "','" + contactInfo + "','" + "null" + "')";
         int result = stmt.executeUpdate(updateStmt);
         stmt.close();
      }
      catch (SQLException sqlex)
      {
         sqlex.printStackTrace();
         System.out.println("mainRegistry error:" + sqlex
         .toString());
         System.out.println("\n: " + updateStmt + "\n");
      }

      // write inputs to the detailed registry
      try
      {
         java.sql.Statement stmt = dbConnection.createStatement();
         String sequence = "INPUT-";
         for ( int i = 0; i < inputClass.size(); i ++ )
         {
            updateStmt = "insert into serviceDetail (" +
                "serviceURL, parameterType, parameterClass" +
                ") values ('" + owlURL + "','" + sequence + i +
                "','" + inputClass.elementAt(i).toString() + "')";
            int result = stmt.executeUpdate(updateStmt);
         }
         stmt.close();
      }
      catch (SQLException sqlex)
      {
         sqlex.printStackTrace();
         System.out.println("serviceDetail error:"+sqlex
         .toString());
         System.out.println("\n: " + updateStmt + "\n");
      }

      // write outputs to the detailed registry
      try
      {
         java.sql.Statement stmt = dbConnection.createStatement();
         updateStmt = "";
         String sequence = "OUTPUT-";
         for ( int i = 0; i < outputClass.size(); i ++ )
         {
            updateStmt = "insert into serviceDetail (" +
                "serviceURL, parameterType, parameterClass" +
                ") values ('" + owlURL + "','" + sequence + i +
                "','" + outputClass.elementAt(i).toString() + "')";
            int result = stmt.executeUpdate(updateStmt);
         }
         stmt.close();
      }
      catch (SQLException sqlex)
      {
         sqlex.printStackTrace();
         System.out.println("serviceDetail error:"+sqlex
         .toString());
         System.out.println("\n: " + updateStmt + "\n");
      }
   }
 }
```

The same is true of the `shutdownDBConnection()` method. Let us move on to the `buildRegistry()` method.

The purpose of `buildRegistry()` includes the following:

- Parse the OWL document and understand whether it is a Semantic Web service description or not.
- If it is indeed a Semantic Web service description file, find all the inputs, outputs, ontologies, and other related information to create an entry in both the `mainRegistry` and `serviceDetail` table.

To accomplish these goals, especially to parse and understand the given OWL document, `buildRegistry()` makes heavy use of Jena APIs. From the previous chapters, we know that Jena APIs is a popular tool for ontology processing, but we did not discuss any examples. With this member function, we finally have a chance to appreciate the power of Jena APIs.

First, `buildRegistry()` creates a Jena model to read in the OWL document (lines 51 to 59). Note that if the OWL document is not well formed, exceptions will be thrown and the processing will end. Therefore, if you want your OWL document to be processed and collected, you should always validate it.

`buildRegistry()` next ensures that the given OWL document is indeed a Semantic Web service description. To do so, `buildRegistry()` checks the following:

- The document has to make use of the `http://www.daml.org/services/owl-s/1.0/Profile.owl` namespace.
- The document has to include either of these two properties:

```
http://www.daml.org/services/owl-s/1.0/Profile.owl#hasInput
http://www.daml.org/services/owl-s/1.0/Profile.owl#hasOutput
```

Lines 60 to 75 in List 14.7 show how this validation is implemented by using Jena APIs. If the given OWL document does pass these checks, it will be assumed to be a Semantic Web service description file.

To further collect all the input and output concepts and other relevant information needed for creating a record for this service, we have to scan the document several times. The first scan is to find the names of the inputs and outputs (lines 82 to 113). Using List 14.2 as an example (the `getCompetitorCamera` service description file), after the first scan the `inputName` vector will have `input-slr` and `input-spec` as its elements and the `outputName` vector will have `output-slr` as its element. Note that the service name and contact information are also collected from the document during the first scan.

The second scan is to find the classes for the input names (lines 114 to 133). By studying the code you can see that the `process:parameterType` is key to accomplishing this. After this scan is done, the `inputClass` vector will have `camera:SLR` and `camera:Specifications` as its elements.

Similarly, the third scan is to find the classes for the output names (lines 134 to 150). This process is similar to the second scan, except that the operation is done for the outputs. After this scan is done, the `outputClass` vector will have `camera:SLR` as its element.

We have now collected all the necessary information to create entries in both the `mainRegistry` and the `serviceDetail` table. The rest of the code is for this purpose. Again, this part is all standard Java programming; I assume you are comfortable with it, and I will not explain it further. After the collected information has been inserted into the tables, you will find the following new records in the `mainRegistry` table:

```
serviceID: http://localhost/chap11/GetCompetitorCameraProfile.owl
ontologyURL: http://www.yuchen.net/photography/Camera.owl
serviceName: getCompetitorCamera
```

```
contactInfo: lyu2@tinman.cs.gsu.edu
WSDLURL: null
```

and the service details are saved in `serviceDetail` table using three records:

```
serviceID: http://localhost/chap11/GetCompetitorCameraProfile.owl
parameterType: INPUT-0
parameterClass: SLR

serviceID: http://localhost/chap11/GetCompetitorCameraProfile.owl
parameterType: INPUT-1
parameterClass: Specifications

serviceID: http://localhost/chap11/GetCompetitorCameraProfile.owl
parameterType: OUTPUT-0
parameterClass: SLR
```

We have now covered the implementation of the crawler and the implementation of building the repository. The next step is to implement the searching functionality, including the matchmaking algorithm discussed earlier. We will cover this last piece in the next section.

14.3.4 Implementation of the Searching Functionalities

We have already accomplished a lot by building the crawler and the repository that holds the descriptions of the Semantic Web services. However, to make this search engine work, some more work needs to be done.

First, we have to build an architecture to hold the entire system. Clearly, the crawler and the repository have to live on the server, and the component that implements the matchmaking algorithm has to be on the server too. On the other hand, there has to be a client that accepts the user request, somehow invokes the search engine, and also presents the search result to the user. Currently, this overall architecture is unclear. Let us discuss this architecture first and then get into the details of the implementation of the searching functionalities.

14.3.4.1 Suggested Architecture for Testing

Before we set up the whole client-server architecture, we have to keep the following points in mind:

- The crawler lives on the server; when invoked by the controller component, it will crawl to the Web to collect the Semantic Web service descriptions and store them in the repository.
- The repository lives on the server; it is supported by a database system (in our case, we are using Microsoft Access as our back-end database system).
- The actual searching component, or the matchmaking engine, resides on the server and has access to the repository.
- The server itself will listen to the incoming calls from the client; once it receives an incoming call, it will invoke the matchmaking engine and send the response (search results) back to the client.

- The client is a Web browser-based client; in other words, the front end of the search engine has the look and feel of an ordinary search engine. It is therefore a very lean client: it is only responsible for accepting user search requests and posting the requests back to the server; the returning results are rendered by the Web browser to the users.
- Given that the client is Web browser based, the incoming calls to the server are `HTTP GET` or `HTTP POST` calls, and responses sent back by the server are in the form of a `HTTP` response stream.
- The components living on the server are implemented using Java.

The server has to be a Web server to satisfy all these criteria. More specifically, this Web server should have the ability to support the HTTP protocol; once it receives an incoming search request from the client, it should invoke some servlets to conduct the search and produce a HTML page that contains the results. Clearly, the matchmaking engine has to be implemented using servlets technology.

Let us now take a closer look at how we can implement such a prototyping system. I assume you will test our Semantic Web service search engine using your development PC either at your home or your office. In other words, there is no dedicated server box created just for testing and prototyping purposes. Therefore, the most practical architecture is to use your development machine as both the server and the client at the same time.

When your development machine is used as the server, `localhost` will be its name. The following steps will set up your `localhost` as a Web server:

1. Download Sun Java System Application Server Platform and install it.
2. Start the server; this will make your `localhost` a Web server capable of handling incoming `HTTP GET` or `HTTP POST` calls; now your `localhost` also supports servlets.
3. Once the server starts, note the port number it is listening to; you need to use this port number to call the server. You can change it if you like.

These steps are listed here to give you an idea of how to set up the architecture needed by our search engine. I will not discuss the details, such as where to download the Sun Java System Application Server, how to install it, or how to configure it (change the port number it uses). You can always follow the instructions on the Sun Web site to perform these tasks.

Now we have finished setting up our server. The next step is to set up the client on the same machine, to complete the testing architecture. This task is fairly simple: all you need to do is to construct a HTML page that takes the user's search request. An example is shown in Figure 14.4. Thus, the client is an HTML page displayed in a Web browser. This page accepts the search request from the user, and once the user clicks the "search it!" button, the browser submits the page to `localhost,` which has been set up already as a Web server listening to some specific port, waiting for incoming client calls.

To ensure that the communication is successful, the port number our `localhost` is listening to has to match the port number that the client page is using. For example,

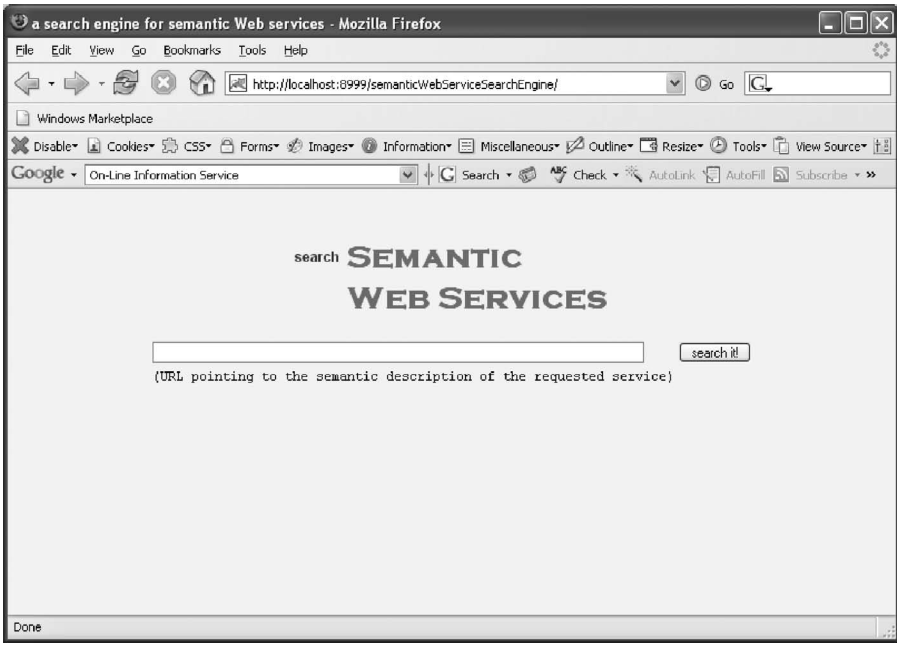

a search engine for semantic Web services - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8999/semanticWebServiceSearchEngine/
Go
Windows Marketplace
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source
Google On-Line Information Service Search Check AutoLink AutoFill Subscribe
search SEMANTIC WEB SERVICES
search it!
(URL pointing to the semantic description of the requested service)
Done

FIGURE 14.4 Client interface of the search engine.

assuming `localhost` is listening to port number 8888, the HTML code of the client page has to include the following line:

```
<form action="http://localhost:8888/myWebServiceSearchEngine"
```

Now `localhost` receives the request; it then directs it to the `myWebServiceSearchEngine` servlet, which does the processing. Once the processing is done, `localhost` returns the resulting HTML page to the client browser; the HTML page is then displayed to the user. This completes the cycle of one search action.

The last piece now is to implement the matchmaking engine, `myWebServiceSearchEngine` servlet. Let us discuss this in the next section.

14.3.4.2 Implementation of the Server-Side Searching Components

As we have discussed, `myWebServiceSearchEngine` servlet is called when `localhost` receives a search request. `myWebServiceSearchEngine` implements the standard `doGet()` and `doPost()` methods and dynamically constructs a HTML page that contains the search result. The search result is obtained by using another class called `myMatchMaker`. The `myWebServiceSearchEngine` servlet does not require much explanation. Let us concentrate on the `myMatchMaker` class, where the real processing is conducted. List 14.8 is the definition of this class.

In order not to make this chapter too long, I have listed only the key member functions of the `myMatchmaker` class.

LIST 14.8
Definition of `myMatchmaker` Class

```
/*
 * myMatchmaker.java
 * Created on April 20, 2006, 3:42 PM
 */
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.rdf.model.*;
import com.hp.hpl.jena.ontology.*;
import com.hp.hpl.jena.shared.PrefixMapping;
import com.hp.hpl.jena.reasoner.*;
import java.util.*;
import java.util.Iterator;

public class myMatchmaker {

   // member variables, download the code from Web site ...

   /** Creates a new instance of myMatchmaker */
   public myMatchmaker(String ns,String url)
   {
      // download the code from Web site ...
   }

   /** Reads the service description document */
   public void readServiceRequest(String requestURL)
   {
  // download the code from Web site ...
   }

   /** Finds all the qualified candidates */
   public void findQualifiedCandidates(Vector allCandidates,
                Vector qualifiedCandidates)
   {
      m = ModelFactory.createOntologyModel
                           (OntModelSpec.OWL_MEM_RULE_INF,null);
      m.read(ontURL);

      for ( int k = 0; k < allCandidates.size(); k ++ )
      {
         // get the service description from this candidate
         myWebServiceDescription sd =
         (myWebServiceDescription)(allCandidates.elementAt(k));

         // if it is not the same domain-specific ontology, skip it
         if ( sd.getONTURL().equalsIgnoreCase(ontURL) == false )
            continue;   // continue to next candidate
          // get input and output from the current candidate
```

```
Vector cInputs = sd.getInputs();
Vector cOutputs = sd.getOutputs();
// if the number of inputs is not same, skip it
if ( cInputs.size() != inputClass.size() )
   continue;  // continue to next candidate
// if the number of outputs is not same, skip
if ( cOutputs.size() != outputClass.size() )
   continue;  // continue to next candidate

// exact match possible?
boolean exactMatch = true;
for ( int i = 0; i < inputClass.size(); i ++ )
{
   if ( cInputs.contains(inputClass.elementAt(i)) ==
        false )
   {
      exactMatch = false;
      break;
   }
}
if ( exactMatch == true )
{
   for ( int i = 0; i < outputClass.size(); i ++ )
   {
      if ( cOutputs.contains(outputClass.elementAt(i))
           == false )
      {
          exactMatch = false;
          break;
      }
   }
}
if ( exactMatch == true )
{
   qualifiedCandidates.addElement(sd);
   continue;  // continue to next candidate
}

// exact match does not exist, other match possible?
boolean inputLevelOneMatch = true;
boolean inputLevelTwoMatch = true;
boolean[] flag = new boolean[inputClass.size()+1];
for ( int j = 0; j < inputClass.size(); j ++ )
   flag[j] = false;

for ( int i = 0; i < inputClass.size(); i ++ )
{
   boolean stopNow = false;
   // the current input could exactly match any of the
   // inputs this candidate has
   for ( int j = 0; j < cInputs.size(); j ++ )
   {
      if ( flag[j] == true ) continue;
      if ( cInputs.elementAt(j).toString().compareTo
           (inputClass.elementAt(i).toString())== 0 )
      {
         flag[j] = true;
         stopNow = true;
         break;
      }
   }
   if ( stopNow == true ) continue;

   // current input cannot exactly match any input of the
   // given candidate. find out if there exist a level-1
   // match for this input
   for ( int j = 0; j < cInputs.size(); j ++ )
   {
      if ( flag[j] == true ) continue;
      if ( isAsuperB(cInputs.elementAt(j).toString(),
           inputClass.elementAt(i).toString()) == true )
      {
         flag[j] = true;
         stopNow = true;
         break;
      }
   }
   if ( stopNow == true )
      inputLevelOneMatch = false;

   // if it reaches here, then at least there exists one
   // input, level-1 match did not work
   for ( int j = 0; j < cInputs.size(); j ++ )
   {
      if ( flag[j] == true ) continue;
      if ( isAsubB(cInputs.elementAt(j).toString(),
           inputClass.elementAt(i).toString()) == true )
      {
         flag[j] = true;
         stopNow = true;
         break;
      }
   }
   if ( stopNow == true ) continue;

   // if it reaches here, then at least there exists one
   // input, level-2 match also failed
   inputLevelTwoMatch = false;
   break;  // there is no need to continue for other inputs
}
```

```
           // if input already fails, no need to check output
           if ( inputLevelOneMatch == false &&
                inputLevelTwoMatch == false )
               continue;  // continue to the next candidate

           // if it reaches here, then the inputs are at least
           // level-2 match. continue to check the outputs
           // similar process, download code from Web site ...

           // now we have finished checking outputs
           if ( outputLevelOneMatch == false
                && outputLevelTwoMatch == false )
              continue;  // continue to the next candidate
            // otherwise this is at least a level-2 matching
            else qualifiedCandidates.addElement(sd);
        }
    }

// decide if the first class (cn1) is a superclass
// of the second one (cn2)
    public boolean isAsuperB(String cn1,String cn2)
    {
       OntClass classA = m.getOntClass(ontNS+cn1);
       OntClass classB = m.getOntClass(ontNS+cn2);
       if ( classA == null || classB == null ) return false;
       return classB.hasSuperClass(classA,false);
    }

// decide if the first class (cn1) is a
// subclass of the second one (cn2)
    public boolean isAsubB(String cn1,String cn2)
    {
       OntClass classA = m.getOntClass(ontNS+cn1);
       OntClass classB = m.getOntClass(ontNS+cn2);
       if ( classA == null || classB == null ) return false;
       return classA.hasSuperClass(classB,false);
    }

  }
```

`findQualifiedCandidates()` is where the discussed matching algorithm is implemented. Parameter `allCandidates` is a vector holding a group of Web service descriptions; each one of them will be studied using the matching algorithm to see whether it matches the requested service. This group of candidates is obtained by reading the repository we built in the last section; the details of that code are not listed here because it is again standard Java programming.

The second parameter passing into the function is called `qualifiedCandidates`. It is empty when the function begins, and it holds the selected candidates when the function ends. Therefore, if it is still empty when the function finishes, the matching algorithm cannot find any candidate that matches the requirement.

Clearly, the matching algorithm will have to use reasoning to finish its task (we have been discussing reasoning power in the earlier chapters in this book), and line 19 creates the ontology model using the parameter `OntModelSpec.OWL_MEM _RULE_INF`. This setting tells Jena to create the ontology model by including not only the base facts into the model (corresponding to the base model) but also all the facts that can be inferred from the base model. Therefore, the created model includes all the facts that might be needed. You may want to study the Jena API documentation to understand more. The ontology here is the ontology in which all the inputs and outputs are defined. In our example, this ontology is the `Camera.owl` ontology we developed.

Lines 21 to 107 show the loop where each candidate service is evaluated to see whether it matches the requirements or not. Line 23 gets the service description details for the current candidate, using the `myWebServiceDescriptions` class. We have not listed this class here. It is a fairly straightforward class, and its instances are created by reading the two tables in the repository; each instance includes all the information we need to implement the matching algorithm.

After getting the service description from the current candidate, we should check if the ontology used by it is the one specified by the user. If not, the current candidate is not a qualified candidate, and we move on to study the next one. Recall that the user has to construct a Semantic Web service description document and specify exactly what he or she is looking for. To do so, the user has to make use of some ontology to describe the semantics of the inputs and outputs in his or her requirement. If the ontology used in the requirement is not the ontology used by the current service candidate, then the current service and the requirement do not share the same semantic context at all, the inputs described in the requirement document and the those specified in the candidate document are not related in any way, and there is no evidence to show that they share a common language. The same is true for the outputs.

If the ontologies match, we continue to check if the number of input parameters required by the current candidate matches that offered by the service requester; we then check the same for the number of output parameters (lines 28 to 31). If one of these two checks fails, we simple ignore the current candidate and move on to the next one.

If all these tests are successful, we then continue to check what kind of mappings can be established between the request and the current candidate. We first try to build an exact mapping between the input parameters. If such a mapping function exists, we then perform the same tests for the output parameters (lines 32 to 51). If tests for both input and output are successful, we know that the current candidate is a perfect match to the request; we therefore add this candidate into the `quali-fiedCandiate` set (line 54) and move on to the next candidate.

If the exact mapping for the input or the output parameters cannot be established, we move on to test if there is a `level-1` or `level-2` mapping for the input parameters. Note that the set of input parameters provided by the service requester is denoted by the `inputClass`, and the set of input parameters required by the current candidate is denoted by `cInputs`. For every input parameter provided by the service requester (line 62), we scan the `cInputs` set to see if any input in the set exactly matches this input parameter (lines 65 to 73). Note that the mapping between `inputClass` and `cInputs` sets has to be a 1-1 mapping; therefore, once a mapping is found, we have to mark the participating element in `cInputs` set as "used," meaning that this specific element in the `cInputs` set has been used in one mapping (`flag[]` array is used for this purpose) and cannot be used in another mapping.

If no exact mapping has been found, we continue to check whether a `level-1` mapping exists (lines 76 to 85). A `level-1` mapping is tested by calling `isAsuperB()`, which returns "true" if class A is a superclass of class B (lines 110 to 116). If this testing is successful for a specific element in `cInputs` set, we know the current input class has a `level-1` mapping to this element; we then mark this element as "used" and move on to the next input class. If this test fails for every element in the `cInputs` set, we know the current input class cannot be mapped to any element in the current candidate input parameter set by using a `level-1` mapping (line 87), and we need to test if this given input can be mapped by using a `level-2` mapping.

`Level-2` testing is done in lines 88 to 98 and is similar to `level-1` testing. If this also fails for the current input parameter, we conclude that even a `level-2` mapping cannot be established using the current candidate (line 99) and there is no further need to test the output parameter sets at all. At this moment, we give up on the current service candidate and move on to the next one (lines 102 and 103).

On the other hand, if the input parameter set has passed the test, meaning that at least a `level-2` mapping has been found, then we continue to test the output parameter sets (line 104). The testing of the output parameter set is similar to the testing of the input parameter set; we are not going to discuss it in detail. Now, if the output test also fails in finding a `level-2` match, we ignore the current candidate and move on (line 105); otherwise, we conclude that this current service can provide at least a level-2 matching. We then add it to the `qualifiedCandidates` set (line 107).

We have now covered the main implementation details. You must have gained a clear understanding of the following topics:

- The architecture of the search engine with the server and client on the same machine
- How to set up the Web server
- The implementation of the crawler
- The implementation of creating the service description repository
- The implementation of the servlet that is responsible for matchmaking

Let us now move on to the last topic of this chapter, a real usage example of our search engine.

14.4 USAGE EXAMPLE OF THE SEMANTIC WEB SERVICE SEARCH ENGINE

In this section, we will look at an example of using the Semantic Web service search engine we just developed. The first task is to take a look at the repository that is created by running the crawler.

14.4.1 Running the Crawler

As discussed earlier, the crawler can be invoked by using the controller, or you can invoke it by using a driver function; it does not matter how you invoke it. Table 14.6 and Table 14.7 show parts of the two tables in the repository created by running the crawler. These tables show some of the service descriptions that I have collected by running the crawler. When you try the crawler, you may find different sets, which is normal; the Internet is a dynamic world and everything is changing all the time. For now, these tables can give you an idea of what information the crawler has collected. Let us move on to the last section of this chapter to see an example of using the search engine.

14.4.2 Querying the Search Engine

To use the search engine we just constructed, assume that we want to search for a service that takes the specifications of a given camera and returns a different camera that offers similar performance. This will be useful if we know a specific camera and its performance but want to find a different (competitor) camera with similar performance so that we can compare the price and other related information.

To do so, we have to express our requirement using a markup language such as OWL-S, and certainly we have to use an ontology to clearly indicate the semantics of the inputs and outputs that we have in mind. Assume that we have studied the `Camera.owl` ontology and believe that it is appropriate for our needs. List 14.9 is a service description we constructed based on the `Camera.owl` ontology. Now start the client we discussed earlier, and enter the URL of the foregoing request document, as shown in Figure 14.5. At this point, you should know what will happen after we hit the "search it!" button. Our `localhost`, working as a Web server, will listen to the port that this request is submitted to; once it gets this request, it will invoke the

TABLE 14.6
The Repository Created by Running the Crawler (`mainRegistry` table)

serviceURL	ontologyURL	serviceName
p1/getCompetitorCameraProfile1.owl	p1/Camera.owl	getCompetitorCamera
p1/getCompetitorCameraProfile2.owl	p1/Camera.owl	getCompetitorCamera
p1/getCompetitorCameraProfile3.owl	p1/Camera.owl	getCompetitorCamera
p1/getMegaPixelProfile.owl	p1/Camera.owl	getMegaPixel
p1/getSpecificationProfile.owl	p1/Camera.owl	getSpecification
p2/abcBookFinder.owl	p3/Concepts.owl	ABC_Books
p2/bravoCarRental.owl	p3/Concepts.owl	Bravo_Car_Rental
p2/congoStockBroker.owl	p3/Concepts.owl	Bravo_Car_Rental
~/dreamInsurance.owl	p3/Concepts.owl	Bravo_Car_Rental
p3/BravoAirProcess.owl	p3/Concepts.owl	null

Note: p1 = `http://tinman.cs.gsu.edu/~lyu2/semanticWeb`; p2 = `http://tinman.cs.gsu.edu/~lyu2/realServices`; p3 = `http://www.daml.org/services/owl-s/1.0`; `contactEmail` and `WSDLURL` are not included in this table.

TABLE 14.7
The Repository Created by Running the Crawler (`serviceDetails` table)

serviceURL	parameterType	parameterClass
p1/getCompetitorCameraProfile1.owl	INPUT-0	SLT
p1/getCompetitorCameraProfile1.owl	INPUT-1	Specifications
p1/getCompetitorCameraProfile1.owl	INPUT-0	SLR
p1/getCompetitorCameraProfile2.owl	OUTPUT-0	Specifications
p1/getCompetitorCameraProfile2.owl	INPUT-0	Digital
p1/getCompetitorCameraProfile2.owl	INPUT-1	Digital
p1/getCompetitorCameraProfile3.owl	INPUT-0	ExpensiveSLR
p1/getCompetitorCameraProfile3.owl	INPUT-1	Specifications
p1/getCompetitorCameraProfile3.owl	OUTPUT-0	ExpensiveSLR
p1/getMegaPixelProfile.owl	INPUT-0	model
p1/getMegaPixelProfile.owl	INPUT-1	pixel
p1/getSpecificationProfile.owl	INPUT-0	SLR
p1/getSpecificationProfile.owl	INPUT-1	Specifications
p2/abcBookFinder.owl	OUTPUT-0	ISBN
p2/abcBookFinder.owl	INPUT-0	string
p2/congoStockBroker.owl	INPUT-0	CompanyTickerSymbol
p2/congoStockBroker.owl	INPUT-1	CreditCard
p2/congoStockBroker.owl	INPUT-2	Integer
p2/congoStockBroker.owl	OUTPUT-0	Stocks
p2/bravoCarRental.owl	INPUT-0	Airport
p2/bravoCarRental.owl	INPUT-1	CarDescription
p2/bravoCarRental.owl	INPUT-2	Integer
p2/bravoCarRental.owl	INPUT-3	RentalDate
p2/bravoCarRental.owl	OUTPUT-0	CarRentalAgreement
p2/dreamInsurance.owl	INPUT-0	VIN
p2/dreamInsurance.owl	OUTPUT-0	CarInsurance
p3/BravoAirProcess.owl	INPUT-0	Password
p3/BravoAirProcess.owl	INPUT-1	Confirmation
p3/BravoAirProcess.owl	INPUT-2	FlightDate
p3/BravoAirProcess.owl	INPUT-3	RoundTrip
p3/BravoAirProcess.owl	INPUT-4	FlightItinerary
p3/BravoAirProcess.owl	INPUT-5	AcctName
p3/BravoAirProcess.owl	INPUT-6	FlightDate
p3/BravoAirProcess.owl	INPUT-7	ReservationNumber
p3/BravoAirProcess.owl	INPUT-8	Airport
p3/BravoAirProcess.owl	INPUT-9	Airport
p3/BravoAirProcess.owl	OUTPUT-0	ReservationNumber
p3/BravoAirProcess.owl	OUTPUT-1	FlightItineraryList
p3/BravoAirProcess.owl	OUTPUT-2	AcctName
p3/BravoAirProcess.owl	OUTPUT-3	FlightItinerary

Note: p1 = `http://tinman.cs.gsu.edu/~lyu2/semanticWeb`; p2 = `http://tinman.cs.gsu.edu/~lyu2/realServices`; p3 = `http://www.daml.org/services/owl-s/1.0`.

LIST 14.9
The Request Document Created Using OWL-S

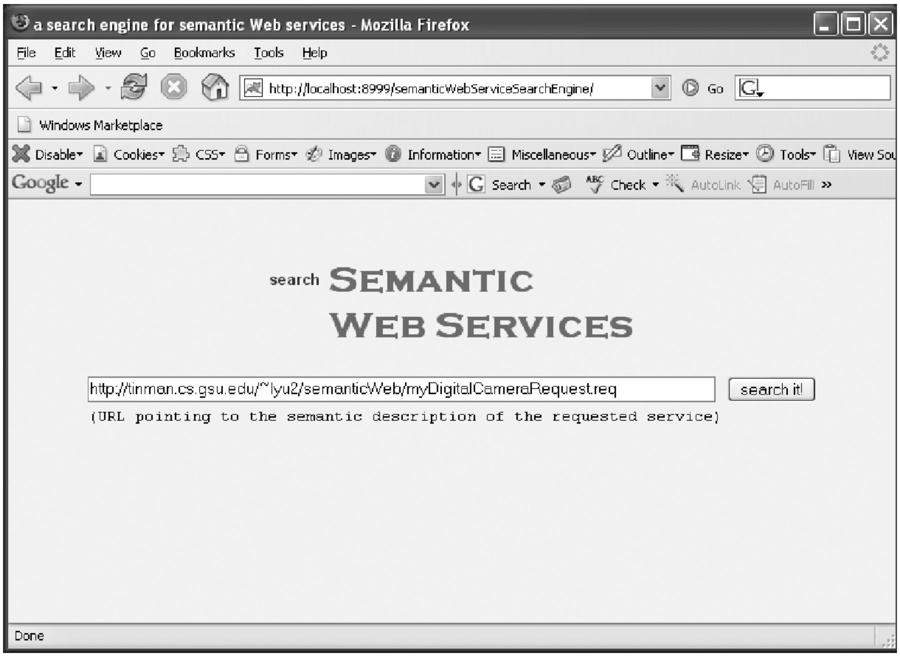
```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE uridef[
 <!ENTITY rdf     "http://www.w3.org/1999/02/22-rdf-syntax-ns">
 <!ENTITY rdfs    "http://www.w3.org/2000/01/rdf-schema">
 <!ENTITY owl     "http://www.w3.org/2002/07/owl">
 <!ENTITY service "http://www.daml.org/services/owl-s/1.0/Service.owl">
 <!ENTITY process "http://www.daml.org/services/owl-s/1.0/Process.owl">
 <!ENTITY profile "http://www.daml.org/services/owl-s/1.0/Profile.owl">
 <!ENTITY actor "http://www.daml.org/services/owl-s/1.0/ActorDefault
               .owl">
 <!ENTITY domainOnt
          "http://tinman.cs.gsu.edu/~lyu2/semanticWeb/Camera.owl">
 <!ENTITY DEFAULT
 "http://tinman.cs.gsu.edu/~lyu2/semanticWeb/myDigitalCameraRequest
 .owl">
]>
<rdf:RDF
   xmlns:rdf=       "&rdf;#"
   xmlns:rdfs=      "&rdfs;#"
   xmlns:owl=       "&owl;#"
   xmlns:service=   "&service;#"
   xmlns:process=   "&process;#"
   xmlns:profile=   "&profile;#"
   xmlns:actor=     "&actor;#"
   xmlns:domainOnt= "&domainOnt;#"
   xmlns=           "&DEFAULT;#">

   <owl:Ontology>
      <owl:imports rdf:resource="&rdf;" />
      <owl:imports rdf:resource="&rdfs;" />
      <owl:imports rdf:resource="&owl;" />
      <owl:imports rdf:resource="&service;" />
      <owl:imports rdf:resource="&profile;" />
      <owl:imports rdf:resource="&process;" />
      <owl:imports rdf:resource="&actor;" />
      <owl:imports rdf:resource="&domainOnt;" />
   </owl:Ontology>

   <profile:Profile rdf:ID="myDigitalCameraRequest">
      <profile:hasInput rdf:resource="#input1-d"/>
      <profile:hasInput rdf:resource="#input2-s"/>
      <profile:hasOutput rdf:resource="#output1-d"/>
   </profile:Profile>
   <process:Input rdf:ID="input1-d">
```

```
      <process:parameterType rdf:resource="&domainOnt;#Digital"/>
   </process:Input>
   <process:Input rdf:ID="input2-s">
      <process:parameterType rdf:resource="&domainOnt;#Specifications
"/>
   </process:Input>
   <process:UnConditionalOutput rdf:ID="output1-d">
      <process:parameterType rdf:resource="&domainOnt;#Digital"/>
   </process:UnConditionalOutput>

</rdf:RDF>
```


a search engine for semantic Web services - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8999/semanticWebServiceSearchEngine/
Go
Windows Marketplace
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Sou
Google Search Check AutoLink AutoFill
search SEMANTIC WEB SERVICES
search it!
Done

http://tinman.cs.gsu.edu/~lyu2/semanticWeb/myDigitalCameraRequest.req

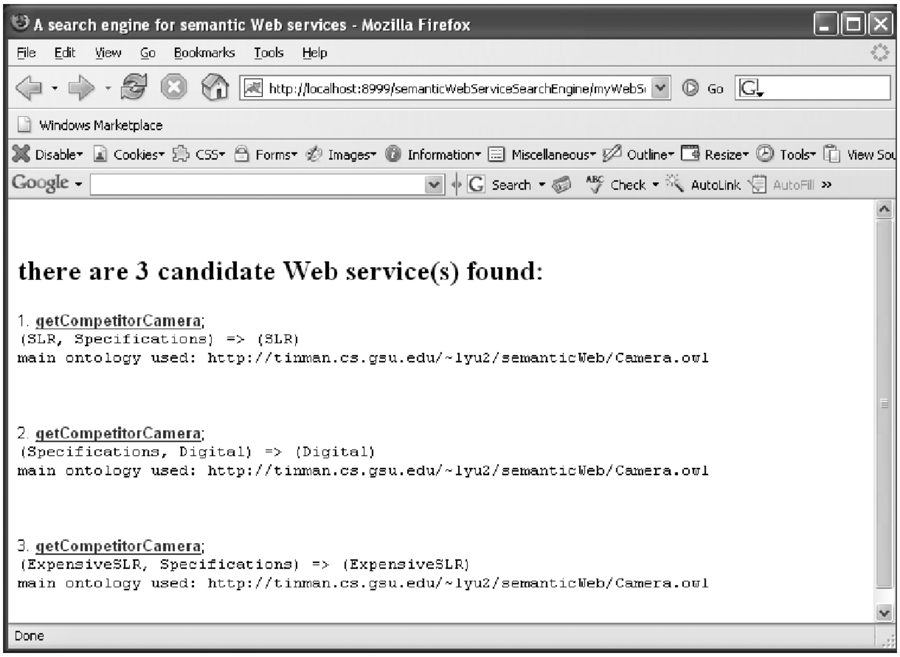
(URL pointing to the semantic description of the requested service)

FIGURE 14.5 Searching for a service using our search engine.

search servlet we developed in the last section and return the results to us when the servlet has finished the search. The result is shown in Figure 14.6.

Let us take a look at the results. The first service candidate requires an `SLR` and `Specifications` as inputs and returns an `SLR` as output. This is a `level-2` match because the input parameter `Digital` is in fact a superclass of `SLR`. Similarly, the other two candidates are all level-2 candidates, and you should be able to explain why. For our specific request, no exact match was found.

At this point, we have finished the book. The last chapter (Chapter 15) presents a brief summary and also points to some future directions.


A search engine for semantic Web services - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8999/semanticWebServiceSearchEngine/myWebS
Go
Windows Marketplace
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Sou
Google
Search Check AutoLink AutoFill
Done

there are 3 candidate Web service(s) found:

1. getCompetitorCamera;
```
(SLR, Specifications) => (SLR)
main ontology used: http://tinman.cs.gsu.edu/~lyu2/semanticWeb/Camera.owl
```

2. getCompetitorCamera;
```
(Specifications, Digital) => (Digital)
main ontology used: http://tinman.cs.gsu.edu/~lyu2/semanticWeb/Camera.owl
```

3. getCompetitorCamera;
```
(ExpensiveSLR, Specifications) => (ExpensiveSLR)
main ontology used: http://tinman.cs.gsu.edu/~lyu2/semanticWeb/Camera.owl
```

FIGURE 14.6 Results returned by our search engine.