6 Validating Your OWL Ontology

After you have developed your ontology and before you set off to do anything exciting (like writing a killer application using some agent), the first question you should ask is, how do I know my OWL ontology is right? Therefore, the first tool you need is a utility that you can use to validate your OWL ontology.

After you have validated that your OWL ontology is right, it then becomes clear that your agent will have to be able to read information from both the instance documents and ontology documents and conduct reasoning based on these documents. So, evidently you need to have some kind of parser and reasoning engine to continue your work and, ideally, these capabilities have to be provided to you in the form of APIs so they are represented in your code base and provide the services you need.

All these capabilities (validation, parsing, reasoning APIs) we just mentioned are part of your Semantic Web development environment. They are tools you can use so you do not have to reinvent the wheel; you can just concentrate on your killer application itself to accomplish your goal of changing the world.

In this chapter, we will discuss some of these tools. We will concentrate on the validation and understanding of ontology documents, not the reasoning power they embody. We will discuss inferencing and reasoning in greater detail in later chapters.

6.1 RELATED DEVELOPMENT TOOLS

What are the development tools available to us? At the time of this writing, we have a very impressive list:

- RDF, RDF schema, and OWL ontology editors: So far, we have covered RDF documents, RDF schema, and OWL ontologies, and we have always created our documents by using a simple editor such as Notepad, or `vi` on a Unix platform. In fact, there are many editors available to make our work easy. You can use these editors to create RDF statements, RDF schemas, or OWL ontologies.

  These editors are just like the editors you have used in different IDEs, for instance, the editor in Visual Studio Integrated Development Environment (IDE). They can offer visual help and check basic syntax on the fly, and they can also export the document in different formats (such as XML, N3, etc.).

  Some examples are listed in Table 6.1 (note that every listing in this chapter will be a partial listing: it is simply not possible to include all the tools here).
- RDF, RDF schema, and OWL ontology Viewer/Browser: These tools offer the capability to visualize classes, properties, and instances, and also provide

a browser-like look and feel. Most editors are also viewers and browsers. For instance, Protégé and Swoop are quite impressive browsers as well.

- RDF, RDF schema, and OWL ontology validator: Now that we have tools to create and view the instance and ontology documents, the next step is to validate these documents. There are quite a few validators available; Table 6.2 only lists a few.
- Web page markup: In Chapter 2 we discussed search engines, and we also mentioned marking up a Web document for the first time. Marking up a document is an extremely important step toward realizing the vision of Semantic Web, and we will talk about markup in much greater detail in later chapters. However, it is amazing that there are many markup tools available in the Semantic Web community. One of these tools is SMORE [38], which creates OWL markup for HTML Web pages.
- RDF, RDF schema, and OWL ontology parsing tools: There are many parsing tools available for use. As every reasoning engine can be used as a parser, to avoid repeating these tools we do not list examples of parsers here. Again, just remember you can use a reasoning engine as a parser; in fact, the very reason why you need to parse some ontology (or instance) documents is to be able to make inferences.
- RDF, RDF schema, and OWL ontology inferencing tools: The next step after validation and parsing is to "understand" the documents. Currently, many tools are available, some of which are in the form of APIs or callable libraries so you can use them in your applications. Table 6.3 lists some of these tools. Again, there are many other inference engines available; we have just named a few as examples.
- RDF, RDF schema, and OWL ontology storage and query: Quite a few inference tools can also be used as tools for storage and query. For instance, every inference engine listed in Table 6.3 is also a good storage and query tool. Another tool worth mentioning is Redland [39]. It is capable of manipulating triples, URIs, and graphs. It also provides a rich API for application development built on top of it. You can get more details about Redland at `http://librdf.org/`.

TABLE 6.1
Tools for Editing Ontology and RDF Documents

Tool Name	Brief Description
Protégé `http://protege.stanford.edu/`	Protégé-OWL editor: create ontology document in a variety of formats
Swoop `http://www.mindswap.org/2004/SWOOP/`	Swoop ontology editor
OilEd `http://oiled.man.ac.uk/`	Editor for ontology documents
RDF Instance Creator `http://www.mindswap.org`	Create RDF statements from ontologies

TABLE 6.2
Tools for Validating Ontology Documents

Tool Name	Brief Description
W3C RDF Validation Service `http://www.w3.org/RDF/Validator/`	RDF document validation provided by W3C; we have used it in Chapter 3
OWL Ontology Validator `http://phoebus.cs.man.ac.uk:9999/OWL/Validator`	OWL ontology validator, quite commonly used
OWL Validator `http://projects.semwebcentral.org/projects/vowlidator/`	OWL validator

TABLE 6.3
Tools for Reasoning Based on Ontology Documents

Tool Name	Brief Description
Jena `http://jena.sourceforge.net/`	Developed by HP; support for RDF, RDF schema, and OWL with reasoning engine
Pellet `http://www.mindswap.org/2003/pellet/`	Open-source Java-based OWL DL reasoner provided by Mindswap
Sesame `http://www.openrdf.org/`	Support for RDF schema reasoning
Euler `http://www.agfa.com/w3c/euler/`	Inference engine supporting logic-based proof

Up to now we have reviewed some popular tools in each category, including editing, validating, parsing, reasoning, and querying. In the rest of this chapter, let us use some of these tools. In the next section, we will take a look at how to use the validation tool to validate our camera ontology, and we will then write Java code to interact with Jean APIs to read and parse our ontology. This should give you a good start as far as development tools are concerned.

6.2 VALIDATE OWL ONTOLOGY BY USING WEB UTILITIES

Let us now validate our camera ontology shown in Chapter 5, List 5.24. The first question to answer is, what exactly do we mean by validating an OWL ontology document?

This validation consists of at least two parts. The first part validates if the syntax is correct. For example, it has to be a legal document in that each opening tag has to have a closing tag. The construct used in the document has to be defined in some namespaces, and the classes and properties mentioned in the document also have to be defined as well. More specifically, if you use something similar to `rdfs:comments`, then the validator will raise a red flag: you should use `rdfs:comment`. A validator may even tell you that you have used a tag `owl:ontology` but your ending

tag is `owl:Ontology`; this case mismatch should be corrected before you can move on. As another example, if you have something like this in your OWL ontology document:

```
<owl:Class rdf:ID="PointAndShoot">
   <rdfs:subClassOf rdf:resource="#Digital"/>
</owl:Class>
```

then it is required that class `Digital` also be defined somewhere in the same document.

The second part of the validation is to validate the semantics. For instance, if you have specified that the `rdfs:range` of a property has to be XML strings, then you cannot use a resource (i.e., some instance) as its value. Also, if you have used a property together with a class or instance, then either that class or any of its superclasses has to be declared related to this property.

Some validators will also output the whole class and property structure based on their own understanding of your document, and if you have created the document right, you should be able to see the class and property structure printed as you wanted. We will see an example of this when using a validator.

6.2.1 Using the "OWL Ontology Validator"

Let us choose the OWL ontology validator developed by Mindswap (`www.mindswap.org`) as our tool to validate the camera ontology. This validator will check what kind of OWL ontology you have by checking it against OWL Full, OWL DL, and OWL Lite. It will tell you the type and also show you the class and property structure found in your document. If there are any syntax errors in your document, it will print out exception messages so you can go ahead and correct the errors.

Let us go to `http://www.mindswap.org/2003/pellet/demo.shtml` to access this validator. Its opening interface is shown in Figure 6.1.

As you can tell, you have two choices when using this validator. You can either cut-and-paste your OWL document into the RDF window or you can specify a URL link that points to your OWL ontology document in the URL textbox. It is normally true that in the development stage, you would prefer not to upload the ontology document to your Web server until it is stable, so let us just cut and paste our document (List 5.24) into the text window. Now set your options by checking the appropriate boxes (not shown in Figure 6.1), and you can start validation by clicking the Submit button.

6.2.2 What the Results Mean

Upon successfully validating the document, the validator returns with the result page shown in Figure 6.2.

We will get this page back only when the submitted camera ontology document is a legal document. If there is something wrong in the document, the validator will throw exceptions so that we can make corrections.

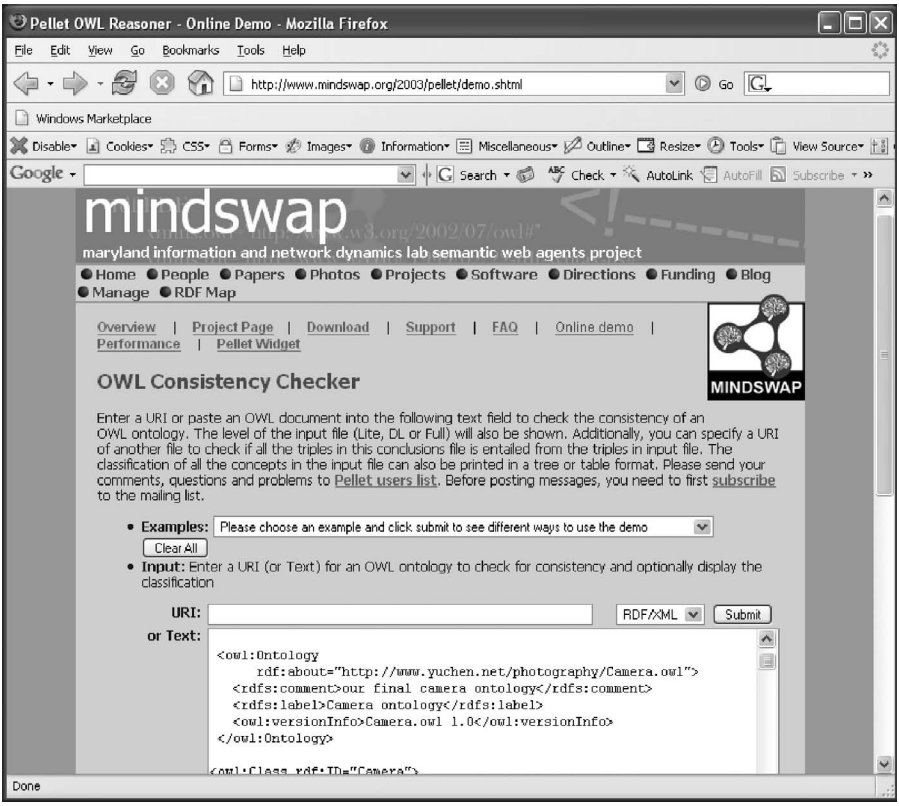

FIGURE 6.1 OWL ontology validator.

The result page shows that the submitted document is an OWL Full version (as we have concluded), and includes the reason for this conclusion. It also outputs the class hierarchy for you to review.

Some other validators may output more information than the one we have just used. Another popular validator is the one developed by the University of Manchester. You can access this validator at this location http://phoebus.cs.man.ac.uk:9999/OWL/Validator. This validator shows a more detailed structure that includes all the classes and properties that we have defined in our ontology. We can examine this structure to further confirm that the ontology does express what we wanted it to express. Let us list the whole structure in List 6.1 so we can take a further look (the line numbers are added for illustrative purposes).

Lines 1 to 7 summarize all the namespaces used in the camera ontology document. The validator also aliases the long namespaces with a much shorter and simpler name.

Line 9 declares the ontology that ends at line 89 — every statement between line 9 and line 89 is part of this ontology. Note that line 9 also identifies the namespace of the underlying ontology.

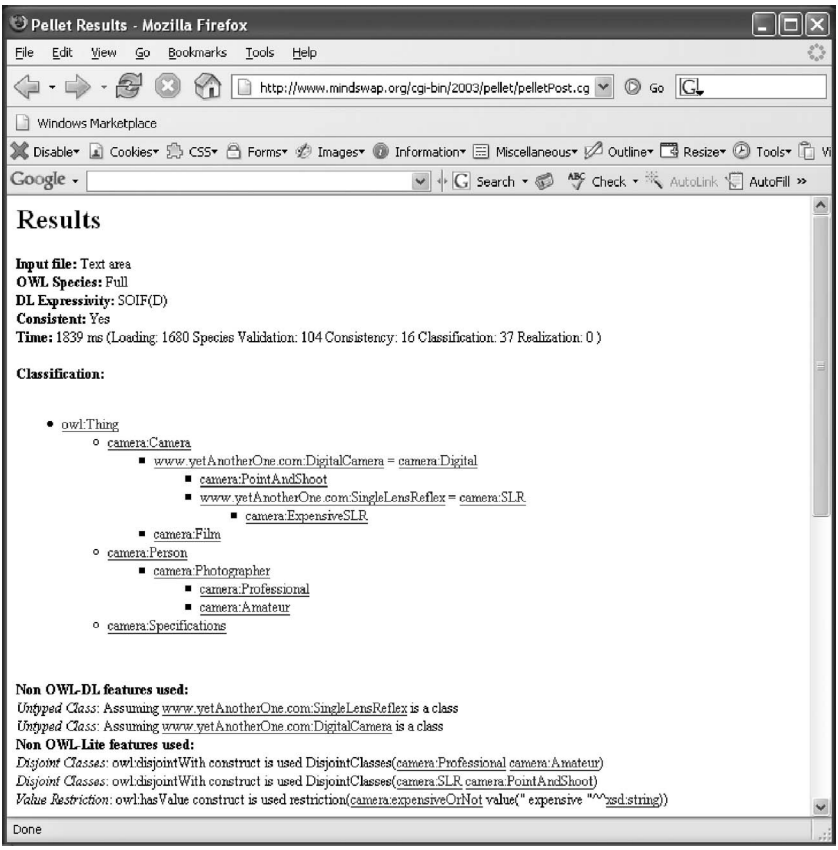

FIGURE 6.2 Validation results.

LIST 6.1
Validation Results from the Validator Developed by the University of Manchester

```
1: Namespace(rdf   = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
2: Namespace(xsd   = <http://www.w3.org/2001/XMLSchema#>)
3: Namespace(rdfs  = <http://www.w3.org/2000/01/rdf-schema#>)
4: Namespace(owl   = <http://www.w3.org/2002/07/owl#>)
5: Namespace(a     = <http://www.yuchen.net/photography/Camera.owl#>)
6: Namespace(b     = <http://www.yetAnotherOne.com#>)
7: Namespace(c     = <http://www.someStandard.org#>)
8:
9: Ontology( <http://www.yuchen.net/photography/Camera.owl>
10:
11: Annotation(rdfs:label "Camera ontology")
12: Annotation(rdfs:comment "our final camera ontology")
13: Annotation(owl:versionInfo "Camera.owl 1.0")
14:
15: ObjectProperty(a:betterQualityPriceRatio Transitive
```

```
16: domain(a:Camera)
17: range(a:Camera))
18: ObjectProperty(a:friend_with Symmetric
19: domain(a:Person)
20: range(a:Person))
21: ObjectProperty(a:has_spec
22: domain(a:SLR)
23: range(a:Specifications))
24: ObjectProperty(a:own
25: inverseOf(a:owned_by)
26: domain(a:Photographer)
27: range(a:SLR))
28: ObjectProperty(a:owned_by
29: inverseOf(a:own)
30: domain(a:SLR)
31: range(a:Photographer))
32:
33: DatatypeProperty(a:expensiveOrNot
34: domain(a:Digital)
35: range(xsd:string))
36: DatatypeProperty(a:model Functional
37: domain(a:Specifications)
38: range(xsd:string))
39: DatatypeProperty(a:pixel
40: domain(a:Digital)
41: range(c:MegaPixel))
42:
43: Class(c:MegaPixel partial
44: xsd:decimal)
45: Class(rdfs:datatype partial)
46: Class(xsd:decimal partial)
47: Class(b:DigitalCamera partial)
48: Class(b:SingleLensReflex partial)
49: Class(a:Amateur partial
50: a:Photographer)
51: Class(a:Camera partial)
52: Class(a:Digital complete
53: b:DigitalCamera)
54: Class(a:Digital partial
55: a:Camera)
56: Class(a:ExpensiveSLR partial
57: restriction(a:owned_by someValuesFrom(a:Professional))
58: a:SLR
59: restriction(a:expensiveOrNot value
      ("expensive"^^<http://www.w3.org/2001/XMLSchema#string>)))
60: Class(a:Film partial
61: a:Camera)
62: Class(a:Person partial)
63: Class(a:Photographer partial
64: a:Person)
```

```
65: Class(a:PointAndShoot partial
66: a:Digital)
67: Class(a:Professional partial
68: a:Photographer)
69: Class(a:SLR complete
70: b:SingleLensReflex)
71: Class(a:SLR partial
72: a:Digital)
73: Class(a:Specifications partial)
74:
75: AnnotationProperty(rdfs:comment)
76: AnnotationProperty(rdfs:label)
77: AnnotationProperty(owl:versionInfo)
78:
79: Individual(c:MegaPixel
80: type(rdfs:datatype))
81: Individual(xsd:string
82: type(rdfs:datatype))
83:
84: DisjointClasses(a:SLR a:PointAndShoot)
85: DisjointClasses(a:Professional a:Amateur)
86:
87: DifferentIndividuals()
88:
89: )
```

Within the ontology description, the first important part is all the `owl:ObjectProperty` and `owl:DatatypeProperty` defined in the ontology (lines 15 to 41).

The class summary is presented from lines 43 to 73. Let us use `ExpensiveSLR` as an example. Lines 56 to 59 state that class `ExpensiveSLR` is a subclass of the following three classes. The first one is an anonymous class whose `owned_by` property has to take an instance of class `Professional` as its value at least once. The second class is the `SLR` class, and the third superclass is another anonymous class whose `expensiveOrNot` value has to be `expensive`. Also, the summary shows that this `expensive` string has to be an XML string by using "`^^<http://www.w3.org/2001/XMLSchema#string>`".

You can read the rest of the summary in a similar way, and as you can tell, our camera ontology is a "good" ontology: its syntax is legal and it expresses exactly what we wanted to express.

Let us move on now to take a look at another way to validate and parse your ontology document: use APIs in your host program.

6.3 USING PROGRAMMING APIS TO UNDERSTAND OWL ONTOLOGY

Another way to validate your OWL document is to use the validation tools programmatically, meaning you have to load and validate a given OWL document in your

main program by calling the APIs provided by these tools. This is important and sometimes becomes necessary. For instance, your agent may discover an OWL file when it visits the Web, and your agent needs to first ensure this is a valid OWL document. It is simply impossible for the agent to stop its work and wait for you to manually validate the document using stand-alone tools. Furthermore, in most cases, the agent has to understand the document and even make inferences based on the knowledge presented in the document.

Many tools provide a programming interface you can use to accomplish this. One of these tools, called Jena [40], is becoming very popular. Let us use Jena as an example to show you how to validate and understand a given OWL document programmatically.

6.3.1 JENA

You can access Jena from http://jena.sourceforge.net/. Developed by HP Labs, Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDF schema, and OWL, including a rule-based inference engine.

Jena is being used in more and more Semantic Web development projects and is included in the tool collections of many Web developers for a simple but important reason: its excellent documentation. In fact, you can find Jena tutorials and examples on its official Web site, and its API document is also quite comprehensive and easy to follow. You can find answers to almost all your questions just by reading their online document.

Using Jena is also straightforward: you just need to download it to your local machine. All you see is a package (`*.jar`) of files that can be used as callable libraries. You do not have to install anything, and you do not actually see any user interface at all.

To use it, you simply use the `import` statement to include the libraries you need. The following two lines are a typical example when you want to use the APIs from Jena:

```
import com.hp.hpl.jena.ontology.*;
import com.hp.hpl.jena.rdf.model.ModelFactory;
```

In order to use the Jena package, you need to ensure that you have set up the `classpath` variable correctly: you have to set up this variable in your configuration file (in both Windows and `UNIX` platforms). List 6.2 is what I have on my Unix server (it is part of my `.cshrc` file).

LIST 6.2
`CLASSPATH` Variable to Set Up Jena Access

```
setenv CLASSPATH $JENA_DIR/lib/antlr.jar:
                 $JENA_DIR/lib/commons-logging.jar:
                 $JENA_DIR/lib/concurrent.jar:
                 $JENA_DIR/lib/icu4j.jar:
                 $JENA_DIR/lib/jakarta-oro-2.0.5.jar:
```

```
$JENA_DIR/lib/jena.jar:
$JENA_DIR/lib/junit.jar:
$JENA_DIR/lib/log4j-1.2.7.jar:
$JENA_DIR/lib/xercesImpl.jar:
$JENA_DIR/lib/xml-apis.jar:.
```

`$JENA_DIR` is the directory on your local machine where you saved your Jena packages, so replace it by using your own path. Remember that you need to add a `classpath` element for every Jena package you want to use; i.e., you have to enumerate every single one, as I did in List 6.2.

Now, let us take a look at how we can use Jena APIs to validate and understand our Camera ontology shown in List 5.24.

6.3.2 Examples

Using Jena to validate our OWL document is straightforward. How do you know your OWL document is valid? Well, if your OWL document is not valid (for example, it may have a syntax or semantic error), Jena will throw exceptions even you just try to load your document into memory using Jena APIs. Reading these exceptions and error messages will normally give you clues about where the problem lies.

On the other hand, if you have successfully used Jena APIs to load the OWL document and created an ontology model, your OWL document is successfully validated. To further confirm this, you can do the following:

- Call Jena API to output the whole ontology.
- Call Jena API to enumerate all the classes you have defined.
- Call Jena API to list all the properties you have defined.

You do not have to do any of the preceding steps if you only want to validate your document; they are mainly provided to satisfy your curiosity or simply for your viewing pleasure.

As we have mentioned, Jena provides you with excellent documentation, including programming examples you can use to do the aforementioned tasks — you can find these coding examples on the Jena Web site. So I am not going to repeat the code, but List 6.3 is part of the camera ontology (List 5.24) printed out by using Jena APIs. One point to note is that after Jena reads the ontology, it changes the ontology document to the long form. List 6.4 is part of the classes outputted by calling Jena APIs.

LIST 6.3
Part of Jena's Output After Reading the Camera Ontology (List 5.24)

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
```

```
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:dc="http://purl.org/dc/elements/1.1/" >

<rdf:Description
    rdf:about="http://www.yuchen.net/photography/Camera.owl#Camera">
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
</rdf:Description>

<rdf:Description
     rdf:about="http://www.yuchen.net/photography/Camera.owl
         #Digital">
    <rdfs:subClassOf
     rdf:resource="http://www.yuchen.net/photography/Camera.owl
         #Camera"/>
   <owl:equivalentClass
        rdf:resource="http://www.yetAnotherOne.com#DigitalCamera"/>
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
</rdf:Description>

<rdf:Description
     rdf:about="http://www.yuchen.net/photography/Camera.owl#SLR">
    <owl:disjointWith rdf:resource="http://www.yuchen.net/photography
         /Camera.owl#PointAndShoot"/>
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
   <owl:equivalentClass
        rdf:resource="http://www.yetAnotherOne.com#SingleLensReflex"/>
   <rdfs:subClassOf
     rdf:resource="http://www.yuchen.net/photography/Camera.owl
         #Digital"/>
</rdf:Description>
```

LIST 6.4
Part of the Class Summary Created by Jena After Reading the Camera Ontology (List 5.24)

```
http://www.yuchen.net/photography/Camera.owl#SLR
http://www.yuchen.net/photography/Camera.owl#PointAndShoot
http://www.yuchen.net/photography/Camera.owl#Specifications
http://www.yuchen.net/photography/Camera.owl#Film
http://www.yuchen.net/photography/Camera.owl#Person
http://www.yuchen.net/photography/Camera.owl#Camera
http://www.yuchen.net/photography/Camera.owl#Photographer
http://www.yuchen.net/photography/Camera.owl#Professional
http://www.yuchen.net/photography/Camera.owl#ExpensiveSLR
http://www.yuchen.net/photography/Camera.owl#Amateur
http://www.yuchen.net/photography/Camera.owl#Digital
http://www.yetAnotherOne.com#SingleLensReflex
http://www.yetAnotherOne.com#DigitalCamera
http://www.someStandard.org#MegaPixel
```

Having all these outputs by using Jena APIs, we can be rest assured that our Camera ontology is valid. In fact, Jena has much more power than just validating a given OWL document, but for now this is good enough for our purposes. We will see much more about Jena's power in later chapters.

Part 3

The Semantic Web: Real-World Examples and Applications

For most of us, learning from examples is an effective as well as efficient way to explore a new subject. In the previous chapters we have learned the core technologies of the Semantic Web. It is time now for some real-world examples and applications.

The chapters in this part will examine two popular Semantic Web examples in great detail: Swoogle and Friend of a Friend (FOAF). Swoogle, as a Semantic Web document search engine, can be quite valuable if you are developing Semantic Web applications or conducting research work in this area. For us too, it is important because it gives us a chance to review what we have learned in the previous chapters, and you will probably be amazed to see there already exist so many ontology documents and RDF instance documents in the real world. FOAF, as a Semantic Web application in the domain of social life, will give you a flavor of using Semantic Web technologies to integrate distributed information over the Internet to generate interesting results. The Semantic Web, to some extent, is all about automatic distributed information processing on a large scale.

This part will also discuss in depth the issue of semantic markup. So far, we have been repeatedly mentioning the idea of "adding semantics to the Web," and as you will see, the process of markup is exactly where this idea translates into action. The rest of the book will also heavily depend on semantic markup as well.

As an example of using the metadata added by semantic markup, we will also revisit the project of building a Semantic Web search engine in this part. In fact, we will design a prototype system whose unique indexation and search process will

show you the remarkable difference between a traditional search engine and a Semantic Web search engine. Given that there is still no "final call" about how a Semantic Web search engine should be built, our goal, therefore, is not only to come up with a possible solution, but also to learn more and appreciate more about the great expectations that have been offered by the vision of the Semantic Web.

Read on.