4 RDFS, Taxonomy, and Ontology

4.1 OVERVIEW: WHY WE NEED RDFS

Congratulations on coming this far with me. Now that we are going to learn a new topic again, let us hear the good news first (and there is no bad news this time): RDFS is written in RDF, so it is not as scary as you might think.

RDFS stands for RDF Schema. In this section, we will talk about why we need RDFS and what it is. As usual, let us go back to our favorite example, shown in List 3.19.

Clearly, it is a perfectly legal RDF document (we have validated this document in the previous chapter): it describes the Nikon D70 camera. But, do you see something is missing here? For example:

- Line 8 says Nikon D70 is an instance of a class called `SLR`, but where is this class defined? What does it look like?
- If `SLR` is a class, are there any other classes that are defined as its superclasses or subclasses?
- The rest of this RDF document describes several properties of this class (such as `soldItem` and `review`), and if you are familiar with object-oriented design, these properties can be viewed as member variables. But the question is, are there any other properties that one can define?

You can ask more questions like these. These questions underline the absence of a vocabulary that defines classes, subclasses, class member variables, and also the relations between these classes.

Yet the reality is that this vocabulary will always be missing in the RDF world. As you can tell, RDF can be used to describe resources in a structured way that machines can process; it can also be used to assert relations between these resources so that machines can be empowered with some basic reasoning capabilities. However, it does not define the vocabulary used; that is, RDF does not say anything about the classes, subclasses and the relations that may exist between these classes.

So, what are the implications if this vocabulary is always missing? Nothing. RDF documents can still be used as a set of stand-alone statements; machines can still read them and make inferences based on these statements. However, this capability will be very limited and can never reach the global level that we are looking for.

To make the distributed information and data over the Internet more machine-friendly and machine-processable, we will need such a vocabulary and, again, we

will have to create this dictionary. As you might have guessed, RDFS is used to create such a vocabulary. It can be viewed as an RDF vocabulary description language. RDFS in conjunction with RDF statements will push the Internet one step further toward machine-readability, and this additional step cannot be accomplished by RDF alone.

What exactly is RDFS, then? We can summarize it as follows:

- RDFS is a language one can use to create a vocabulary for describing classes, subclasses, and properties of RDF resources; it is a recommendation from W3C [14].
- The RDFS language also associates the properties with the classes it defines.
- RDFS can add semantics to RDF predicates and resources: it defines the meaning of a given term by specifying its properties and what kinds of objects can be the values of these properties.

As we have mentioned before, RDFS is written in RDF. In fact, not only is RDFS written in RDF, but RDFS also uses the same data model as RDF, i.e., a graph or triples. In this sense, RDFS can be viewed as an extension of RDF.

Before we get into the details of RDFS, let us see how it can help us by making the Internet more machine-processable. This is dealt with in the next section.

4.2 RDFS + RDF: ONE MORE STEP TOWARD MACHINE-READABILITY

As discussed in the previous section, RDFS is all about vocabulary. To see the power of such a vocabulary, let us build one first; see Figure 4.1. It is a simple vocabulary, but it is good enough to demonstrate our point, and it will become richer in the later sections.

This simple vocabulary tells us the following fact:

> We have a resource called `Camera`, and `Digital` and `Film` are its two subresources. Also, resource `Digital` has two subresources, `SLR` and `Point-and-Shoot`. Resource `SLR` has a property called `has-spec`, whose value is the resource called `Specifications`. Also, `SLR` has another property called `owned-by`, whose value is the resource `Photographer`, which is a subresource of `Person`.

Now consider the RDF document in List 4.1. Together with the vocabulary shown in Figure 4.1, what inferences can be made by the machine? Actually, it is quite impressive; see List 4.2. In fact, I did not list all the conclusions that can be drawn by the machine. Try to list the rest of these conclusions yourself. For example, the machine can also conclude that `http://www.yuchen.net/rdf/NikonD70#Nikon-D70` is also a `Digital` resource, right?

Note that the foregoing reasoning is not done by you or me; it is done by a machine. Think about this for a while, and you should see that given the structure of RDF triples and given the structure of the vocabulary (we will see these structures in the next section), it is not difficult for a machine to carry out this reasoning. The final result is that a machine seems to be able to understand the information on the Web.

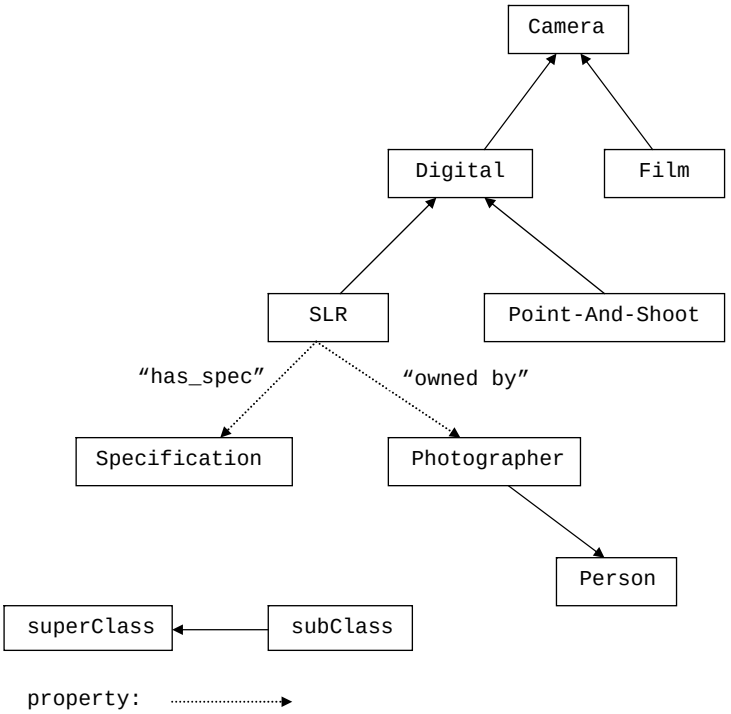

FIGURE 4.1 A simple camera vocabulary.

LIST 4.1
A Simple RDF Document Using the Vocabulary Shown in Figure 4.1

```
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:          xmlns="http://www.yuchen.net/photography/Camera.rdfs#">
4:   <rdf:Description
       rdf:about="http://www.yuchen.net/rdf/NikonD70.rdf#Nikon-D70">
5:       <rdf:type
          rdf:resource="http://www.yuchen.net/photography/Camera#SLR"/>
6:         <owned_by rdf:resource="http://www.yuche.net/people#Liyang
            Yu"/>
7:   </rdf:Description>
8: </rdf:RDF>
```

Can this vocabulary help search engines? Yes. Recall the Semantic Web search engine presented in Chapter 2; I leave it to you to come up with the reasons why this vocabulary can help this search engine. Here is a hint: Suppose I use the keyword `Digital` to search for information about digital cameras; assume also that one Web document has been marked up to have the aforementioned RDF triples associated with it. The search engine reads the RDF triples and concludes that this Web page

LIST 4.2
Conclusions Drawn by the Machine Based on List 4.1 and Figure 4.1

```
Fact:          "owned-by" is a property used to describe SLR and now it
               is used to describe http://www.yuchen.net/rdf/
               NikonD70#Nikon-D70
conclusion:    http://www.yuchen.net/rdf/NikonD70#Nikon-
               D70 must be a SLR
Fact:          Only Photographer can be used as the value of property
               "owned-by" and now http://www.yuchen.net/people#LiyangYu
               is used as its value
conclusion:    http://www.yuchen.net/people#LiyangYu must be a
               Photographer
Fact:          Photographer is also a Person
Conclusion:    http://www.yuchen.net/people#LiyangYu is a Person
```

is about `SLR`, and as `SLR` is a subresource of `Digital`, this Web document is indeed relevant; the search engine will include this page into its returned page set. Note that this Web page may not contain the word `Digital` at all!

The important point is that by using just RDF triples, the foregoing reasoning cannot be done; the power comes from the combination of RDF triples and the RDF vocabulary, which we call RDF schema.

We now have enough motivation to dive into the details of RDF schema, which is about how to express the vocabulary (such as the one shown in Figure 4.1) in such a way that the machine can understand.

4.3 CORE ELEMENTS OF RDFS

4.3.1 Syntax And Examples

In this section, the following core elements will be discussed:

Core classes: `rdfs:Resource`, `rdf:Property`, `rdfs:Class`, `rdfs:datatype`
Core properties: `rdfs:subClassOf`, `rdfs:subPropertyOf`
Core constraints: `rdfs:range`, `rdfs:domain`

Let us start by defining the resource `Camera`, the top resource in Figure 4.1. Note that "resource" in the world of RDF schema has the same semantics as "class," so these two words will be used interchangeably. Also, let us name the rdfs document `Camera.rdfs`. List 4.3 shows what we do when defining a class in the RDF schema file.

LIST 4.3
Using RDFS to Define the `SLR` Class

```
//
// Camera.rdfs
```

```
//
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:          xml:base="http://www.yuchen.net/photography/Camera.rdfs">
5:    <rdf:Description rdf:ID="Camera">
6:      <rdf:type
          rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
7:   </rdf:Description>
8: </rdf:RDF>
```

Let us understand List 4.3 line by line. First of all, everything is defined between `<rdf:RDF>` and `</rdf:RDF>`, indicating this document is either an RDF document or an RDF schema document. Starting from line 2, several namespaces are declared. The new one here is the `rdfs` namespace; the keywords for the RDF schema are defined in this namespace. Line 4 defines a namespace for our camera vocabulary. Note that `xml:base` is used; therefore, the namespace does not end with #, as the rule of concatenating the URI is as follows:

```
xml:base + # + rdf:ID value
```

Lines 5 to 7 define the class `Camera` using `rdf:ID` and `rdf:type` properties. We can interpret these lines as follows:

> A class, `Camera`, is defined in this RDF schema document; it is a subclass of `rdfs:Resource`.

Note that if we define a class without specifying any `rdfs:subClassOf` property (explained later), it is then assumed that this defined class is a subclass of `rdfs:Resource`, which is the root class of all classes.

You might have guessed that we have a simpler way of expressing the same idea. You are right. Using the simpler form, the `Camera.rdfs` file now looks like the one in List 4.4. This simpler form (which does not use the `rdf:type` property) looks much cleaner and is more intuitive: line 5 uses `rdfs:Class` to define a class, and it also uses `rdf:ID` to provide a name, `Camera`, to the newly defined class. We will use this form for the rest of this book.

LIST 4.4
A Simpler Version of List 4.3

```
//
// Camera.rdfs
//
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:          xml:base="http://www.yuchen.net/photography/Camera.rdfs">
5:   <rdfs:Class rdf:ID="Camera">
```

```
6:   </rdfs:Class>
7: </rdf:RDF>
```

Again, every class is assumed to be a subclass of `rdfs:Resource`. `Camera` is used as a top-level class in our vocabulary, so it is a direct subclass of `rdfs:Resource`. Another top class in our vocabulary is `Person`, and we can also define it in the same RDF schema document; see List 4.5.

LIST 4.5
Adding Class `Person` to List 4.4

```
//
// Camera.rdfs
//
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:          xml:base="http://www.yuchen.net/photography/Camera.rdfs">
5:   <rdfs:Class rdf:ID="Camera">
6:   </rdfs:Class>
7:   <rdfs:Class rdf:ID="Person">
8:   </rdfs:Class>
9: </rdf:RDF>
```

What about the subclasses in the vocabulary? For instance, `Digital` is a subclass of `Camera`. How are these subclasses defined? We can use the `rdfs:subClassOf` property to ensure the class is a subclass of the other class. This is shown in List 4.6.

LIST 4.6
Adding Subclasses

```
//
// Camera.rdfs
//
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:          xml:base="http://www.yuchen.net/photography/Camera.rdfs">
5:   <rdfs:Class rdf:ID="Camera">
6:   </rdfs:Class>
7:   <rdfs:Class rdf:ID="Person">
8:   </rdfs:Class>
9:   <rdfs:Class rdf:ID="Digital">
10:     <rdfs:subClassOf rdf:resource="#Camera"/>
11:  </rdfs:Class>
12:  <rdfs:Class rdf:ID="Film">
```

```
13:     <rdfs:subClassOf rdf:resource="#Camera"/>
14:  </rdfs:Class>
15:  <rdfs:Class rdf:ID="SLR">
16:     <rdfs:subClassOf rdf:resource="#Digital"/>
17:  </rdfs:Class>
18:  <rdfs:Class rdf:ID="Point-And-Shoot">
19:     <rdfs:subClassOf rdf:resource="#Digital"/>
20:  </rdfs:Class>
21:  <rdfs:Class rdf:ID="Photographer">
22:     <rdfs:subClassOf rdf:resource="#Person"/>
23:  </rdfs:Class>
24:  <rdfs:Class rdf:ID="Speicifications">
25:  </rdfs:Class>
26: </rdf:RDF>
```

Lines 9 to 25 define all the subclasses that are used in our vocabulary shown in Figure 4.1. The key RDF schema property used to accomplish this is `rdfs:subClassOf`. Let us study this property in greater detail.

First, note how the base class is identified in the `rdfs:subClassOf` property. For instance, line 9 defines a class, namely, `Digital`, and line 10 uses the `rdfs:subClassOf` property to specify that the base class of `Digital` is `Camera`. `Camera` is identified as follows:

```
<rdfs:subClassOf rdf:resource="#Camera"/>
```

This is perfectly fine in this case as when the parser sees `#Camera`, it assumes that class `Camera` must have been defined in the same document (which is true in this case). To get the URI of class `Camera`, it concatenates `xml:base` and this name together to get the following:

```
http://www.yuchen.net/photography/Camera.rdfs#Camera
```

This is clearly the right URI for this class. But what if the base class is defined in some other document? The solution is simple: use the full URI for the class. We will see such examples later.

Second, `rdfs:subClassOf` can be used multiple times to describe a class. Let us say you have already defined a class `Artist`; you can define `Photographer` as follows:

```
<rdfs:Class rdf:ID="Photographer">
  <rdfs:subClassOf rdf:resource="#Person"/>
  <rdfs:subclassOf rdf:resource="#Artist"/>
</rdfs:Class>
```

This means class `Photographer` is a subclass of both `Person` class and `Artist` class. Therefore, any instance of `Photographer` is simultaneously an instance of both `Person` and `Artist`. What if `rdfs:subClassOf` property is not used at all, as when we defined class `Camera`? Then any instance of `Camera` is also an instance of class `rdfs:Resource`.

> Up to this point, we have covered the following RDF schema vocabulary: `rdfs:Class` and `rdfs:subClassOf`.

Now that we have defined all the classes in our camera vocabulary, let us define properties.

To define a property, `rdf:Property` type is used. The `rdf:ID` in this case specifies the name of the property; furthermore, `rdfs:domain` and `rdfs:range` together indicate how the property is being defined. Let us take a look at List 4.7.

LIST 4.7
Adding Property Definitions

```
//
// Camera.rdfs
//
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:          xml:base="http://www.yuchen.net/photography/Camera.rdfs">
5:
... // all the classes definitions as shown in version 0.3
25:
26:  <rdf:Property rdf:ID="has_spec">
27:     <rdfs:domain rdf:resource="#SLR"/>
28:     <rdfs:range rdf:resource="#Specifications"/>
29:  </rdf:Property>
30:  <rdf:Property rdf:ID="owned_by">
31:     <rdfs:domain rdf:resource="#SLR"/>
32:     <rdfs:range rdf:resource="#Photographer"/>
33:  </rdf:Property>
34: </rdf:RDF>
```

As shown in List 4.7, lines 26 to 29 define the property called `has_spec`, and lines 30 to 33 define another property called `owned_by`. Using `has_spec` as an example, we can interpret this as follows:

> We define a property called `has_spec`. It can only be used to describe the characteristics of class (domain) SLR, and its possible values (range) can only be instances of class `Specifications`.

Or equivalently:

- Subject: `SLR`
- Predicate: `has_spec`
- Object: `Specifications`

You can use the same idea to understand the property owned_by defined in lines 30 to 33. Let us discuss rdfs:domain and rdfs:range.

RDF schema property rdfs:domain is used to specify which class the property being defined can be used with (read the previous example again). It is optional; in other words, you can declare property owned_by like this:

```
<rdf:Property rdf:ID="owned_by">
    <rdfs:range rdf:resource="#Photographer"/>
</rdf:Property>
```

This declaration indicates that property owned_by can be used to describe any class; for instance, you can say "a Person is owned_by a Photographer"! In most cases, this is not what you want, so you may create your definition like this:

```
<rdf:Property rdf:ID="owned_by">
   <rdfs:domain rdf:resource="#SLR"/>
   <rdfs:range rdf:resource="#Photographer"/>
</rdf:Property>
```

Now it makes sense: owned_by property can only be used with the class specified by rdfs:domain, namely, SLR.

It is interesting to know that when you define a property, you can specify multiple rdfs:domain properties. In this case, you are indicating that the created property can be used with an instance that is an instance of every class defined by rdfs:domain property. For example,

```
<rdf:Property rdf:ID="owned_by">
   <rdfs:domain rdf:resource="#SLR"/>
   <rdfs:domain rdf:resource="#Point-And-Shoot"/>
   <rdfs:range rdf:resource="#Photographer"/>
</rdf:Property>
```

This states that the property owned_by can only be used with something that is an SLR camera and a point-and-shoot camera at the same time. In fact, an SLR camera can be used as a point-and-shoot camera, so you can say that an SLR camera is also a point-and-shoot camera.

The same scenarios can be used with rdfs:range. First of all, it is optional, like the following:

```
<rdf:Property rdf:ID="owned_by">
   <rdfs:domain rdf:resource="#SLR"/>
</rdf:Property>
```

This states that property owned_by can be used with SLR class, but it can take any value. You have seen the case where exactly one rdfs:range property is used (previous example). When you use multiple rdfs:range properties such as this (assume we have also defined a class called Journalist):

```
<rdf:Property rdf:ID="owned_by">
   <rdfs:domain rdf:resource="#SLR"/>
   <rdfs:range rdf:resource="#Photographer"/>
   <rdfs:range rdf:resource="#Journalist"/>
</rdf:Property>
```

This states that property owned_by can be used to depict SLRs, and its value has to be someone who is a Photographer and Journalist at the same time: a photojournalist.

Before we move on to the next topic, there are several points you should pay attention to. Let us look at these now.

The first point is that Class is in the rdfs namespace and Property is in the rdf namespace. Therefore, there is no typo in the preceding lists.

Second, when defining the aforementioned two properties, we used the abbreviated form. It is important to know this as you might see the long form in other documents. The two forms are equivalent (see List 4.8).

LIST 4.8
Short Form vs. Long Form

```
Shortform:
<rdf:Property rdf:ID="owned_by">
   <rdfs:domain rdf:resource="#SLR"/>
   <rdfs:range rdf:resource="#Photographer"/>
</rdf:Property>
Longform:
<rdf:Description rdf:ID="owned_by">
   <rdf:type
        rdf:resource="http://www.w3.or/1999/02/22-rdf-syntx-
            ns#Property"/>
   <rdfs:domain rdf:resource="#SLR"/>
   <rdfs:range rdf:resource="#Photographer"/>
</rdf:Description>
```

The last point to be aware is that when we use rdfs:domain and rdfs:range properties, the rdf:resource is always written similarly to the following (see the previous lists):

```
<rdfs:domain rdf:resource="#SLR"/>
```

Again, the reason we can do this is because SLR is declared locally; i.e., SLR is defined in the same document. If we need to use some resource that is not defined locally, we need to use the proper URI for that resource. For instance, suppose that Journalist is defined in some other namespace called http://someOtherNs.org instead of locally, we should use the following:

```
<rdf:Property rdf:ID="owned_by">
   <rdfs:domain rdf:resource="#SLR"/>
   <rdfs:range rdf:resource="#Photographer"/>
   <rdfs:range rdf:resource="http://www.someOtherNs.org#Journalist"/>
</rdf:Property>
```

Up to this point, we have covered the following RDF schema vocabulary: rdfs:Class, rdfs:subClassOf, rdf:Property, rdfs:domain, and rdf:range.

As we discussed earlier, property `rdfs:range` is used to specify the possible values of a property being declared. In some cases, the property being defined can simply have an untyped string as its value. For example, if we define a property called `model` in our simple camera vocabulary, and this property can take values such as `D70` (a simple string), we can declare it like this:

```
<rdf:Property rdf:ID="model">
   <rdfs:domain rdf:resource="#Specifications"/>
   <rdfs:range
         rdf:resource="http://www.w3.org/2001/01/rdf-schema
             #Literal"/>
</rdf:Property>
```

In fact, you can even omit the `rdfs:range` specification, indicating that `model` property can take any value:

```
<rdf:Property rdf:ID="model">
   <rdfs:domain rdf:resource="#Specifications"/>
</rdf:Property>
```

However, the problem with using an untyped string or any value is that the agent is deprived of reasoning power; this will become clearer in the later sections, but you should be able to see the reason for this: if you can use anything as the value, how can an inference engine make any judgment at all?

Therefore, a better idea is to always provide typed values if you can. For example, we can specify that the valid value for the `model` property has to be strings specified in the XML schema; see List 4.9.

LIST 4.9
Example of Using Typed Value

```
1: <rdf:Property rdf:ID="model">
2:    <rdfs:domain rdf:resource="#Specification"/>
3:    <rdfs:range rdf:resource="&xsd;string"/>
4: </rdf:Property>
5: <rdfs:datatype rdf:about="&xsd;string"/>
```

Line 3 specifies that the property `model` takes values of type `xsd:string` (the full URI is `http://www.w3.org/2001/XMLSchema#string`). We can use this URI directly in our schema without explicitly indicating that it represents a data type. However, it is always useful to clearly declare that a given URI represents a data type. This is done in line 5.

The next example shows that using `rdfs:datatype` is not only good practice, but is also necessary in some cases. For instance, let us assume we add a `pixel` property to the class `Digital`, one of the most important thing about a digital camera being its `pixel` value (a measure of the quality of the picture). See List 4.10.

LIST 4.10
Property and its Datatype Definition

```
1: <rdf:Property rdf:ID="pixel">
2:    <rdfs:domain rdf:resource="#Digital"/>
3:    <rdfs:range rdf:resource="http://www.someStandard.org#MegaPixel"/>
4: </rdf:Property>
5: <rdfs:datatype rdf:about="http://www.someStandard.org#MegaPixel">
6:    <rdfs:subClassOf
            rdf:resouce="http://www.w3.org/2001/XMLSchema#decimal"/>
7: </rdfs:datatype>
```

When an RDF schema parser reaches line 3, it concludes that the values `pixel` (a property of class `Digital`) can take should come from a resource with `http://www.someStandard.org#MegaPixel` as its URI. However, when it reads line 5, it realizes this URI identifies a data type, and `http://www.w3.org/2001/XMLSchema#decimal` is the base class of this data type. Therefore, the parser concludes that `pixel` should always use a typed literal as its value.

Note that when we use `rdfs:datatype` in our RDF schema document to indicate a data type, the corresponding RDF instance statements must use `rdf:datatype` property as follows:

```
<model rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
       D70</model>
<pixel rdf:datatype="http://www.someStandard.org#MegaPixel”>6.1
       </pixel>
```

Now that we have two new properties added to our PDF schema document, let us update Figure 4.1. This gives a new version of our vocabulary, as shown in Figure 4.2.

Note that in Figure 4.2, class `Digital` has a property called `pixel`, and also two subclasses, namely, `SLR` and `Point-And-Shoot`. Then, do these subclasses also have the property `pixel`? The answer is yes, and the rule is: subclasses always inherit properties from their base class. Therefore, classes `SLR` and `Point-And-Shoot` both have a property called `pixel`.

In fact, a class always inherits properties from all the base classes it has. For instance, class `Camera` is also a base class of `SLR`, and if we assume `Camera` has a property called `manufactured_by`, then `SLR` will have two properties inherited from its two superclasses: `pixel` and `manufactured_by`.

We can also define a property to be a subproperty of another property. This is done by using `rdfs:subPropertyOf`. For example, the `model` property describes the official name of a camera. However, the manufacturer could sell the same model using different model names. For instance, a camera sold in North America could have a different model name when it is sold in Asia. Therefore, we can define another property, say, `officialModel`, to be a subproperty of `model`:

```
<rdf:Property rdf:ID="officialModel">
    <rdfs:subPropertyOf rdf:resource="#model"/>
</rdf:Property>
```

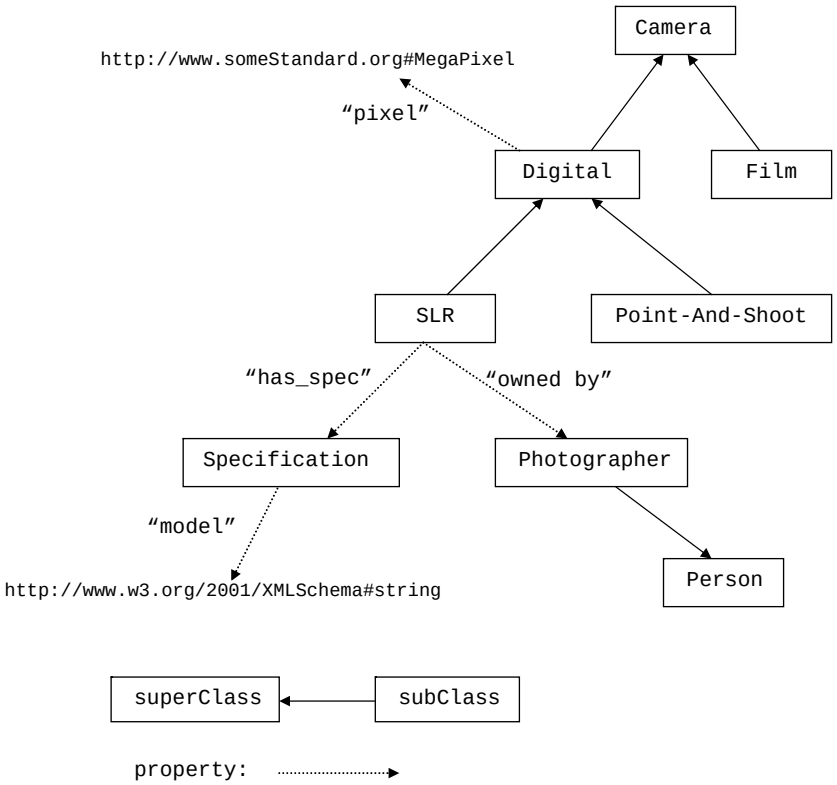

FIGURE 4.2 A simple camera vocabulary (new properties added).

This declares that the property `officialModel` is a specialization of property `model`. Property `officialModel` inherits `rdfs:domain` and `rdfs:range` values from its base property `model`. However, you can narrow the domain or the range as you wish.

Again, when we define a property, we can use the `rdfs:subPropertyOf` property for different cases. If we define a property without using `rdfs:subPropertyOf`, we are creating a top-level property. If we use `rdfs:subPropertyOf` once (as shown in the previous example), we are indicating that the property being defined is a specialization of another property. If we decide to use multiple `rdfs:subPropertyOf`, we are declaring that the property being defined has to be a subproperty of each of the base properties.

Up to now, we have covered the most important classes and properties in RDF schema. The last two properties you may encounter in documents are `rdfs:label` and `rdfs:comment`. The former is used to provide a class or property name for humans and, similarly, `rdfs:document` provides a human-readable description of the property or class being defined. One example is shown in List 4.11.

Another issue is the usage of `rdfs:XMLLiteral`. I recommend that you avoid using it, and here are the reasons. First, `rdfs:XMLLiteral` denotes a well-formed XML string, and it is always used together with `rdf:parseType="Literal"`; if

LIST 4.11
Example of Using `rdfs:label` and `rdfs:comment`

```
1: <rdf:Property rdf:ID="officialModel">
2:    <rdfs:subPropertyOf rdf:resource="#model"/>
3:    <rdfs:label xml:lang="EN">officialModelName</rdfs:label>
4:    <rdfs:comment xml:lang="EN">this is the official name of the
            camera. the manufacturer may use different names when
            the camera is sold in different regions/countries.
5:    </rdfs:comment>
6: </rdf:Property>
```

you use `rdfs:XMLLiteral` in an RDF schema document to define some property, the RDF statements that describe an instance of this property will have to use `rdf:parseType="Literal"`. Let us see an example. Suppose we want to define a new property called features:

```
<rdf:Property rdf:ID="features">
   <rdfs:domain rdf:resource="#Digital"/>
   <rdfs:range
    rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-
        ns#XMLLiteral"/>
</rdf:Property>
```

Therefore, the property `features` is used with `Digital` class, and its value can be any well-formed XML. An example RDF statement could be:

```
<features rdf:parseType="Literal">
   Nikon D70 is <bold>good!</bold>, also, ...
</features>
```

Note that you need to use `rdf:parseType="Literal"` to indicate this is well-formed XML content.

Although the content is well-formed XML, in general it does not have the `resource`, `property`, and `value` structure. And as you have already seen, this structure is one of the main reasons why tools and agents can "understand" the content. Therefore, if we use XML paragraph as the value of some property, no tool will be able to understand the meaning well. So, avoid using `XMLLiteral` if you can.

A reminder: you can still use `rdfs:Literal`; it is an untyped literal (string) and can be useful in some cases.

> Up to this point, we have covered the following RDF schema vocabulary: `rdfs:Class`, `rdfs:subClassOf`, `rdf:Property`, `rdfs:domain`, `rdf:range`, `rdfs:datatype`, `rdfs:subPropertyOf`, `rdfs:label`, and `rdfs:comment`.

4.3.2 More about Properties

Now that we have established a good understanding of RDF schema, let us discuss some deeper issues related to properties.

You might have already noticed that properties are defined separately from classes. Those who are used to the object-oriented world might find this fact uncomfortably strange. For instance, in any object-oriented language (such as Java or C++), you would define a class called `DigitalCamera`, and then encapsulate several properties to describe a digital camera. These properties will be defined when you define the class, and they are defined in the class scope. In other words, they are considered to be member variables owned by the class being defined and they are local to the `DigitalCamera` class, not directly visible to the outside world.

For the RDF schema, it is quite a different story. You define a class, and very often you also indicate its relationships with other classes. However, this is it: you never declare its members, i.e., the properties it may have. For you, a class is just an entity that has relationships with other existing entities. What is inside this entity, i.e., its member variables and properties, are unknown.

Actually, you declare properties separately and associate them with classes if you wish to do so. In other words, properties are never owned by any class, they are never local to any class, and if you do not associate a given property with any class, this property is independent and can be used to describe any class.

The key question now is, what is the reason behind this? What is the advantage of separating the class and property definitions? Before reading on, think about it; you should be able to figure out the answer by now.

The answer is Rule #3 that we discussed in Chapter 3. Let me repeat it here:

> Rule #3: The most exciting one!
>
> I can talk about resource at my will, and if I chose to use an existing URI to identify the resource I am talking about, then the following is true:
>
> 1. The resource I am talking about and the resource already identified by this existing URI represent exactly the same resource.
> 2. Everything I have said about this resource is considered to be additional knowledge about that resource.

We have already seen why this rule is important: it makes the distributed information spread all over the Internet machine-processable. A hypothetical example is presented in the previous chapter to show that such an application is possible.

Back to the world of RDF schema: the separation of the class and property definitions is just an implementation of this rule. The final result is that the agent or tool we use will have more power to automatically process the distributed information, together with a stronger inferencing engine.

For instance, someone else could create another RDF schema document with a new property defined, say, `aperture`, and associate it with our `SLR` class by using `http://www.yuchen.net/photography/Camera.rdfs#SLR` as its URI. This is an implementation of rule #3. Anyone, anywhere, and anytime can talk about a resource by adding more properties to it. Now an automatic agent can collect all these statements distributed over different Web pages, and its reasoning power is enhanced. I leave it to you to come up with an example to show why reasoning power is enhanced with this extra knowledge. Clearly, if the definitions of class and property were not separate, this would not have been possible.

The next feature of property is not as exciting as the aforementioned one, but it is an important programming trick you should know. Let us modify the `owned_by` property as follows:

```
<rdf:Property rdf:ID="owned_by">
   <rdfs:domain rdf:resource="#Digital"/>
   <rdfs:domain rdf:resource="#Film"/>
   <rdfs:range rdf:resource="#Photographer"/>
</rdf:Property>
```

If we define `owned_by` property like this, we are asserting that `owned_by` is to be used with instances that are simultaneously digital cameras and film cameras. Clearly, such a camera has not been invented yet. Actually, we wanted to express the fact that a photographer can own a digital camera, or film camera, or both. How do we accomplish this?

Given that a subclass will inherit all the properties associated with its base class, we can associate the `owned_by` property with the base class:

```
<rdf:Property rdf:ID="owned_by">
   <rdfs:domain rdf:resource="#Camera"/>
   <rdfs:range rdf:resource="#Photographer"/>
</rdf:Property>
```

As both `Digital` and `Film` are subclasses of `Camera`, they all inherit the `owned_by` property. Now we can use the `owned_by` property with the `Digital` class or the Film class, and the problem is solved.

4.3.3 XML Schema and RDF Schema

We have discussed the relationship between RDF and XML in Chapter 3, and later in this book you will see more and more reasons why XML alone is not enough to make the Semantic Web vision a reality. An equally important question is the relationship between XML and RDF schemas. We will cover this topic in this section.

First of all, the purpose of XML schema is to validate an XML document; i.e., to ensure its syntax is legal. It accomplishes this by defining the allowed structure and data types of an XML document. List 4.12 is a simple XML schema.

LIST 4.12
Simple XML Schema Example

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3c.org/2001/XMLSchema">
<xsd:element name = "DigitalCamera">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name = "model" type = "xsd:string"
                   maxOccurs = "unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Based on the preceding schema, a valid XML document would be similar to the one shown in List 4.13.

LIST 4.13
An XML Document Based on the Schema Described in List 4.12

```
<?xml version="1.0" encoding="UTF-8"?>
<DigitalCamera
    xmlns:xsi = "http://www.w3c.org/2001/XMLSchema-Instance"
    xsi:noNameSpaceSchemaLocation = "http://www.Dot.com/mySchema.xsd">
  <model>Nikon D70</model>
  <model>Dikon D50</model>
  <model>Canon EOS 20D</model>
</DigitalCamera>
```

This is pretty much all there is to say about XML schema. Again, it is all about syntax and is intended to validate an XML instance document created by following the syntax specified by the XML schema document.

On the other hand, RDF schema is an extension of RDF; it provides the vocabulary that can be used by the RDF instance statements. It defines the classes and the relationships between them. It also defines properties and associates them with the classes. The final result is a vocabulary that can be used to describe knowledge, and it has nothing to do with validation in any sense.

Going one step further, RDF schema is all about semantics. By now you should realize how semantics is expressed in RDF schemas. Let us again summarize it using the following two important points:

- Semantics, or the meaning of a given term, is defined by specifying its properties and what kinds of objects can be the values of these properties.
- Semantics can be understood by a machine by following the structure of "`resource-property-propertyValue.`"

As long as these design guidelines are adhered to, an automatic and large-scale agent can be constructed to help accomplish some really exciting goals.

4.4 THE CONCEPTS OF ONTOLOGY AND TAXONOMY

4.4.1 What is Ontology?

Now seems to be the correct moment to talk about the concept of *ontology*. The truth is, we have already built one; the vocabulary in Figure 4.2 is an ontology.

There are many definitions of ontology, and perhaps each of these views ontology from a different perspective. In the world of the Semantic Web, let us use the operational definition of ontology from W3C's OWL Requirements Documents (you will learn all about Web Ontology Language (OWL) in the Chapter 5):

An ontology defines the terms used to describe and represent an area of knowledge. [33]

There are several aspects of this definition that need to be clarified. First, this definition states that ontology is used to describe and represent an area of knowledge. In other words, ontology is domain specific; it is not there to represent all knowledge, but an area of knowledge. A domain is simply a specific subject area or sphere of knowledge, such as photography, medicine, real estate, education, etc.

Second, ontology contains terms and the relationships among these terms. Terms are often called classes, or concepts; these words are interchangeable. The relationships between these classes can be expressed by using a hierarchical structure: superclasses represent higher-level concepts and subclasses represent finer concepts, and the finer concepts have all the attributes and features that the higher concepts have.

Third, besides the aforementioned relationships among the classes, there is another level of relationship expressed by using a special group of terms: properties. These property terms describe various features and attributes of the concepts, and they can also be used to associate different classes together. Therefore, the relationships among classes are not only superclass or subclass relationships, but also relationships expressed in terms of properties.

To summarize, an ontology has the following properties:

- It is domain specific.
- It defines a group of terms in the given domain and the relationships among them.

By clearly defining terms and the relationships among them, an ontology encodes the knowledge of the domain in such a way that it can be understood by a computer. This is the basic purpose of an ontology.

In the world of the Semantic Web, you may encounter another concept: *taxonomy*. Taxonomy and ontology are quite often used interchangeably; however, they are different concepts. As discussed earlier, ontology defines not only the classes but also their properties. It further indicates the type of values these properties may have and what classes they may be associated with, thereby creating sophisticated relationships among the classes. On the other hand, taxonomy mainly concerns itself with classification issues — not the properties — to express further constraints and relationships. For instance, if we get rid of all the properties defined in Figure 4.2, it will be termed a taxonomy and not an ontology.

4.4.2 Our `Camera` Ontology

As we mentioned earlier, RDF schema is a language for building ontologies, and we already built one during the course of this chapter. It is expressed in Figure 4.2 and from now on, we will call it the *camera ontology*.

The current version of the camera ontology is shown in List 4.14. This is our first ontology using RDF schema, and it is expressed in Figure 4.2 in graphical form.

LIST 4.14
Our Camera Ontology Using RDFS

```
//
// Camera.rdfs
// our camera ontology
//
1: <?xml version="1.0"?>
2: <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3:          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4:          xml:base="http://www.yuchen.net/photography/Camera.rdfs">
//
// classes definitions
//
5:    <rdfs:Class rdf:ID="Camera">
6:   </rdfs:Class>
7:   <rdfs:Class rdf:ID="Person">
8:   </rdfs:Class>
9:   <rdfs:Class rdf:ID="Digital">
10:     <rdfs:subClassOf rdf:resource="#Camera"/>
11:  </rdfs:Class>
12:  <rdfs:Class rdf:ID="Film">
13:     <rdfs:subClassOf rdf:resource="#Camera"/>
14:  </rdfs:Class>
15:  <rdfs:Class rdf:ID="SLR">
16:     <rdfs:subClassOf rdf:resource="#Digital"/>
17:  </rdfs:Class>
18:  <rdfs:Class rdf:ID="Point-And-Shoot">
19:     <rdfs:subClassOf rdf:resource="#Digital"/>
20:  </rdfs:Class>
21:  <rdfs:Class rdf:ID="Photographer">
22:     <rdfs:subClassOf rdf:resource="#Person"/>
23:  </rdfs:Class>
24:  <rdfs:Class rdf:ID="Speicifications">
25:  </rdfs:Class>
//
// property definitions
//
26:  <rdf:Property rdf:ID="has_spec">
27:     <rdfs:domain rdf:resource="#SLR"/>
28:     <rdfs:range rdf:resource="#Specifications"/>
29:  </rdf:Property>
30:  <rdf:Property rdf:ID="owned_by">
31:     <rdfs:domain rdf:resource="#SLR"/>
32:     <rdfs:range rdf:resource="#Photographer"/>
33:  </rdf:Property>
34:  <rdf:Property rdf:ID="model">
35:     <rdfs:domain rdf:resource="#Specification"/>
36:     <rdfs:range
```

```
            rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
37:  </rdf:Property>
38:  <rdfs:datatype rdf:about="http://www.w3.org/2001/XMLSchema
          #string"/>
39: <rdf:Property rdf:ID="pixel">
40:    <rdfs:domain rdf:resource="#Digital"/>
41:    <rdfs:range rdf:resource="http://www.someStandard.org
            #MegaPixel"/>
42: </rdf:Property>
43: <rdfs:datatype rdf:about="http://www.someStandard.org#MegaPixel">
44:    <rdfs:subClassOf
            rdf:resouce="http://www.w3.org/2001/XMLSchema#decimal"/>
45: </rdfs:datatype>
46: </rdf:RDF>
```

4.4.3 The Benefits of Ontology

We can summarize the benefits of ontology as follows (and you should be able to come up with most of these benefits yourself):

- It provides a common and shared understanding/definition about certain key concepts in the domain.
- It provides a way to reuse domain knowledge.
- It makes the domain assumptions explicit.
- Together with ontology description languages (such as RDF schema), it provides a way to encode knowledge and semantics such that machines can understand.
- It makes automatic large-scale machine processing possible.

Among all these benefits, we in fact pay more attention to the fourth one in the preceding list. To convince ourselves about these exciting benefits, let us take another look at our camera ontology to see how it can make our machine more intelligent. This will lead us to the next section.

In the next section, not only you will see more reasoning power provided by our camera ontology, but you will also find aspects that can be improved. This points to another new building block called OWL, the details of which will be presented in Chapter 5.

4.5 ANOTHER LOOK AT INFERENCING BASED ON RDF SCHEMA

4.5.1 Simple, Yet Powerful

Earlier in this chapter, we used an example to show you how reasoning is done by using the camera ontology (we called it camera vocabulary then). In this section, we present this reasoning ability in a more formal way, together with the extra reasoning examples that we did not cover in the previous section.

With the help of the camera ontology, a smart agent can accomplish reasoning in the following ways:

1. Understand a resource's class type by reading the property's `rdfs:domain` tag: When we define a property **P**, we normally use `rdfs:domain` to specify exactly which class this property **P** can be used to describe; let us use **C** to denote this class. Now, for a given resource with a specific URI, if the agent detects that property **P** is indeed used to describe this resource, the agent can then conclude that the resource represented by this particular URI must be an instance of class **C**. An example of this type of reasoning was presented in the previous section.
2. Understand a resource's class type by reading the property's `rdfs:range` tag: When we define a property **P**, we normally use `rdfs:range` to specify exactly what are the possible values this property can assume. Sometimes, this value can be a typed or untyped literal, and sometimes it has to be an instance of a given class **C**. Now, when parsing a resource, if the agent detects that property **P** is used to describe this resource, and the value of **P** in this resource is represented by a specific URI pointing to another resource, the agent can then conclude that the resource represented by this particular URI must be an instance of class **C**. An example of this type of reasoning was presented in the previous section, too.
3. Understand a resource's superclass type by following the class hierarchy described in the ontology: This can be viewed as extension of the preceding two reasoning scenarios. In both these cases, the final result is that the class type of some URI (resource) is successfully identified. Now the agent can scan the class hierarchy defined in the ontology. If the identified class has one or more superclasses defined in the ontology, then the agent can conclude that this particular URI is not only an instance of this identified class, but also instance of all the superclasses. Again, an example of this reasoning was also presented in an earlier section.
4. Understand more about the resource by using the `rdfs:subPropertyOf` tag: Let us use an example to illustrate this reasoning. Suppose we have defined the following property:

```
<rdf:Property rdf:ID="parent">
   <rdfs:domain rdf:resource="#Person"/>
   <rdfs:range rdf:resource="#Person"/>
</rdf:Property>
<rdf:Property rdf:ID="mother">
   <rdfs:subClassOf rdf:resouce="#parent"/>
</rdf:Property>
```

This defines two properties, namely, `parent` and `mother`; also, `mother` is a subproperty of `parent`. Assume we have a resource in the RDF statement document:

```
<Person rdf:ID="Liyang">
   <mother>
      <Person rdf:about="#Zaiyun"/>
```

```
    </mother>
</Person>
```

When parsing this statement, the agent realizes that `Liyang`'s `mother` is `Zaiyun`. To go one step further, as `mother` is a subproperty of `parent`, it then concludes that `Liyang`'s parent is also `Zaiyun`. This can be very useful in some cases.

The foregoing examples are the four main ways in which agents can make inferences based on the given ontology (and certainly the instance file). These are indeed simple, yet very powerful already, and made possible mainly by the following:

- All this reasoning power is made possible by having an ontology defined.
- The `resource-property-propertyValue` structure always ensures that reasoning can be conducted in an efficient way, even on a large scale.

4.5.2 Good, Better and Best: More is Needed

Although RDF schema is quite impressive already, there are still gaps in it. For example, what if we have two classes representing the same concept? More precisely, we have an `SLR` class in our camera ontology; if there is another ontology that uses `Single-Lens-Reflex` as the class name, these two classes would represent the same concept. Reasoning power would be greatly enhanced if we could somehow indicate that these two classes are equivalent. However, using RDF schema, it is not possible to accomplish this.

Another obvious example is that there are no cardinality constraints available using RDF schema. For example, `pixel` is a property that is used to describe the image size of a digital camera and for a particular camera, there is only one `pixel` value. However, in your RDF document, you can use as many as `pixel` properties on a given digital camera.

Therefore, there is indeed a need to extend RDF schema to allow for the expression of complex relationships among classes and the precise constraints on specific classes and properties. Further, we need a more advanced language that will be able to perform the following functions, among others:

- Express relationships among classes defined in different documents across the Web
- Construct new classes by unions, intersections, and complements of other existing classes
- Add constraints on the number and type for properties of classes
- Determine if all members of a class will have a particular property, or if only some of them might

This new language is called OWL, and it is the main topic of Chapter 5.