Parsing Nested XML

James W. Cooper

Many years ago, when I was young and charming, as some of you may know I practiced baby bean farming. But right there in the small patch in my wide-open backyard I found that the tops were being nibbled off every bean plant as soon as it came of age. I solved the problem one day while hoeing between rows, when I heard a loud squeaking noise. At first I thought I'd hit a child's toy, but upon deeper consideration, I discovered I was hoeing into a nest of baby rabbits!

Well this nest wasn't too hard to deal with, but much more recently I received some patent documents in XML format, and needed to writ e a little parser to pull out a few of the more important fields. There was a single file with a whole series of concatenated patents, having the form shown below.

```
<document form='VTDIM'>
<item name='PN'><text>WO0226988</text></item>
<item name='PUD'><text>2002-04-04</text></item>
<item name='TTL'><text>DRUG METABOLIZING ENZYMES</text></item>
<item name='INV'><text>DUGGAN BRENDAN M; BOROWSKY MARK L </text></item>
<item name='APD'><text>2001-09-28</text></item>
<item name='SPEC' sign='true' seal='true'>
      <richtext>
            <par>DRUG METABOLIZING ENZYMES </par>
      </richtext>
</item>
<item name='ACLM' sign='true' seal='true'>
      <richtext>
            <par>What is claimed is: 1. </par>
      </richtext>
</item>
<item name='ABST' sign='true' seal='true'>
      <richtext>
      <par>The invention provides human drug</par>
      </richtext>
</item>
</document>
```

While there are a large number of other fields, the important ones we want to extract are

PN- patent number

TTL- title

INV- inventors

APD- application date

SPEC- specifications

ACLM- claims

ABST- abstract

Parsing XML with a SAX Parser

You will probably recall that there are two types of XML parsers: DOM parsers and SAX parsers. A DOM parser parses the entire XML file at once and reads it into memory. A SAX parser triggers calls to methods in an org.xml.sax.helpers.DefaultHandler class each time an XML tag is read. It is up to you to interpret the tags and to catch the text between the tags and save the data or build a tree as needed. This latter SAX approach is the one we'll use here, because it is scalable to multiple document files of any size.

In particular, you need to derive your own handler class from DefaultHandler to override the following methods:

startElement, endElement – an XML element has started or ended

startDocument, endDocument – a document has started or ended.

characters – the characters between the tags are caught in this method.

The trouble with this summary of those methods is that it seems to imply that there can only be one such set of methods and that you must find out which tag occurred and make appropriate decisions within each method. This would lead to terribly convoluted code and would be contrary to OO programming methods.

Instead, we'll build a little class for each kind of tag and switch between them using a factory.

The Base Tag parser class

We start by creating a base tag parsing class that has default implementations of the startElement, endElement and characters mthods.

```
/**
 * base class for all tag parsing classes
 */
public class TagParser extends DefaultHandler {
      protected StringBuffer buffer;//characters accumulate here
      protected String name, value; //first attribute name and value
      protected String tagName;     //name of tag
      protected Mediator med;       //mediator

      /** Constructor for TagParser  */
      public TagParser(String tagName, Mediator med) {
            super();
            this.tagName = tagName;
            this.med = med;
            buffer = new StringBuffer(); //create a new buffer
      }
      //------
      /** all characters between tags accumulate here
      * may be called any number of times between tags.
      * **/
      public void characters(char[] ch, int start, int length)
            throws SAXException {
            //append all chars to buffer
            buffer.append(ch, start, length);
      }
      //------
```

```
        //a tag has been started
        public void startElement( //.. .

        }
}
```

Reading Attributes

Now most of the XML tags in this example have one or more *attributes* that are contained inside the tag. For example, the first attribute inside the item tag defines the type of tag:

```
<item name='PN'>
```

We take care of these attributess in the complete startElement method as shown below. In this simple case, we save the name and value of the first attribute:

```
//a tag has been started
        public void startElement(
                String uri,
                String localName,
                String qName,
                Attributes attrib)
                throws SAXException {
                //create a new character buffer
                buffer = new StringBuffer();

                //see if there are any attributes
                int length = attrib.getLength();
                if (length > 0) { //if there are save the first one
                        name = attrib.getQName(0);
                        value = attrib.getValue(0);
                }
}
```

The Program's Objective

Now, what we want to do is to write a program to pick out some of the more important fields in each patent and then dump them to text files for analysis by other programs. We will create a PatentDocument object with getters and setters for each of the fields we are interested in ad a PatentWriter class to write out the text. The PatentDocument class has the form

```
public class PatentDocument {
        private String title;   //document title
        private String id;      //document ID

        private String claims;  //claims text
        private String abst;    //abstract
        private String spec;    //specification
        private String path;    //path where we write out the file
        //-----
        public PatentDocument(String path) {
                this.path = path;
                title = id = claims = abst = spec = "";
```

```
        }
```

and has getters and setters for each of the private variables.

The real issue here is how do we know which tag we are currently parsing, so we know which field in the PatentDocument we want to update. We could use a bunch of *if* tests and flags, but we will instead use a Factory, a Mediator and a Template Method pattern.

Nested XML

The whole secret to parsing nested XML is to create parse handlers for each tag and switch them into place when that tag begins and restore the previous parse handler when that tag ends. We use the Mediator class to handle the switching between these parse handlers. The complete class diagram is shown in Figure 1. At the outset, it creates instances of handlers for each of the tags we are interested in and stores them in a hash table.

```
public Mediator(String inputFile, String path) {
            parsers = new Hashtable();
            stack = new Stack();
            this.path = path;
            parsers.put("PN", new PnumberParser(this));
            parsers.put("TTL", new TitleParser(this));
            parsers.put("SPEC", new SpecParser(this));
            parsers.put("ACLM", new ClaimParser(this));
            parsers.put("ABST", new AbsParser(this));
            parseHandler = new DocumentParser(this);
            try {
                  SAXParserFactory sFact =
SAXParserFactory.newInstance();
                  SAXParser sParser = sFact.newSAXParser();
                  File fl = new File(inputFile);
                  sParser.parse(fl, this);
            } catch (SAXException e) {
                  System.out.println(e.getMessage());
                  e.printStackTrace() ;
            } catch (ParserConfigurationException e) {
                  System.out.println(e.getMessage());
            } catch (IOException e) {
                  System.out.println(e.getMessage());
            }
      }
```

The Mediator itself becomes the SAX parser and passes on the calls to the current parse handler, keeping previous ones on a stack.

```
//a tag has been started
      public void startElement(
            String uri,
            String localName,
            String qName,
            Attributes attrib)
            throws SAXException {

            String name, value = "";
            //get the document parser if this is a document tag
```

```
        if (qName.equals("document")) {
              //save old parser
              stack.push( parseHandler);
              parseHandler = new DocumentParser(this);
        }
        //choose one of the item parsers
        //if this is an item tag
        if (qName.equals("item")) {
              //save old parser
              stack.push(parseHandler);
              int length = attrib.getLength();
              //get the first attribute
              if (length > 0) {
                    name = attrib.getQName(0);
                    value = attrib.getValue(0);
              }
              //get the new parse handler
              //from the hash table
              parseHandler = getParser(value);
        }
        //pass call to new parse handler
        parseHandler.startElement(uri, localName, qName, attrib);
  }
```

When a tag ends, it calls the endElement method, which in turn calls the endElement method of the current parseHand ler:

```
//a tag has ended
      public void endElement(String uri,
            String localName, String qName)
            throws SAXException {
            parseHandler.endElement(uri, localName, qName);
      }
```

This is the critical piece because it is at that point that the data are copied in to the right field of the document. For example for the PN (Patent Number) field, the PnumberParser does the following:

```
public void endElement(String uri,
      String localName, String qName)
            throws SAXException {
                  if(qName.equals(tagName) ){
                        copyBuffer();
                        med.endElement(); //pops handler off stack
                  }
}
//-----
protected void copyBuffer() {
      med.getDocument().setId( buffer.toString().trim() );
}
```

Then, back in the Mediator, the previous parseHandler is restored:

```
public void endElement() {
      parseHandler = (TagParser) stack.pop();
}
```

Writing the ParseHandlers

Each parseHandler has a startElement and endElement method and a copyBuffer method that copies the data to the right place in the PatentDocument object. We start by creating an abstract class with an abstract copyBuffer method, which forces all the derived classes to implement one specifically:

```
public abstract class ItemParser extends TagParser {
	public ItemParser(Mediator med) {
		super("item", med);
	}
	//a tag has started
	public void startElement(
		String uri,
		String localName,
		String qName,
		Attributes attrib)
			throws SAXException {
				super.startElement( uri, localName,
					qName, attrib);
	}
	//a tag has ended
	public void endElement(String uri,
		String localName, String qName)
		throws SAXException {
			if(qName.equals(tagName) ){
				copyBuffer();
				med.endElement(); //pops handler off stack
			}
	}
	//copy the results somewhere
	protected abstract void copyBuffer() ;
}
```

This is actually most of the code. The derived handlers really just have to implement the copyBuffer method to put the data accumulated by the characters() method somewhere. Here is the complete parseHandler for patent numbers

```
public class PnumberParser extends ItemParser {
	public PnumberParser(Mediator med) {
		super(med);
	}

	protected void copyBuffer() {
		med.getDocument().setId( buffer.toString().trim() );
	}
}
```

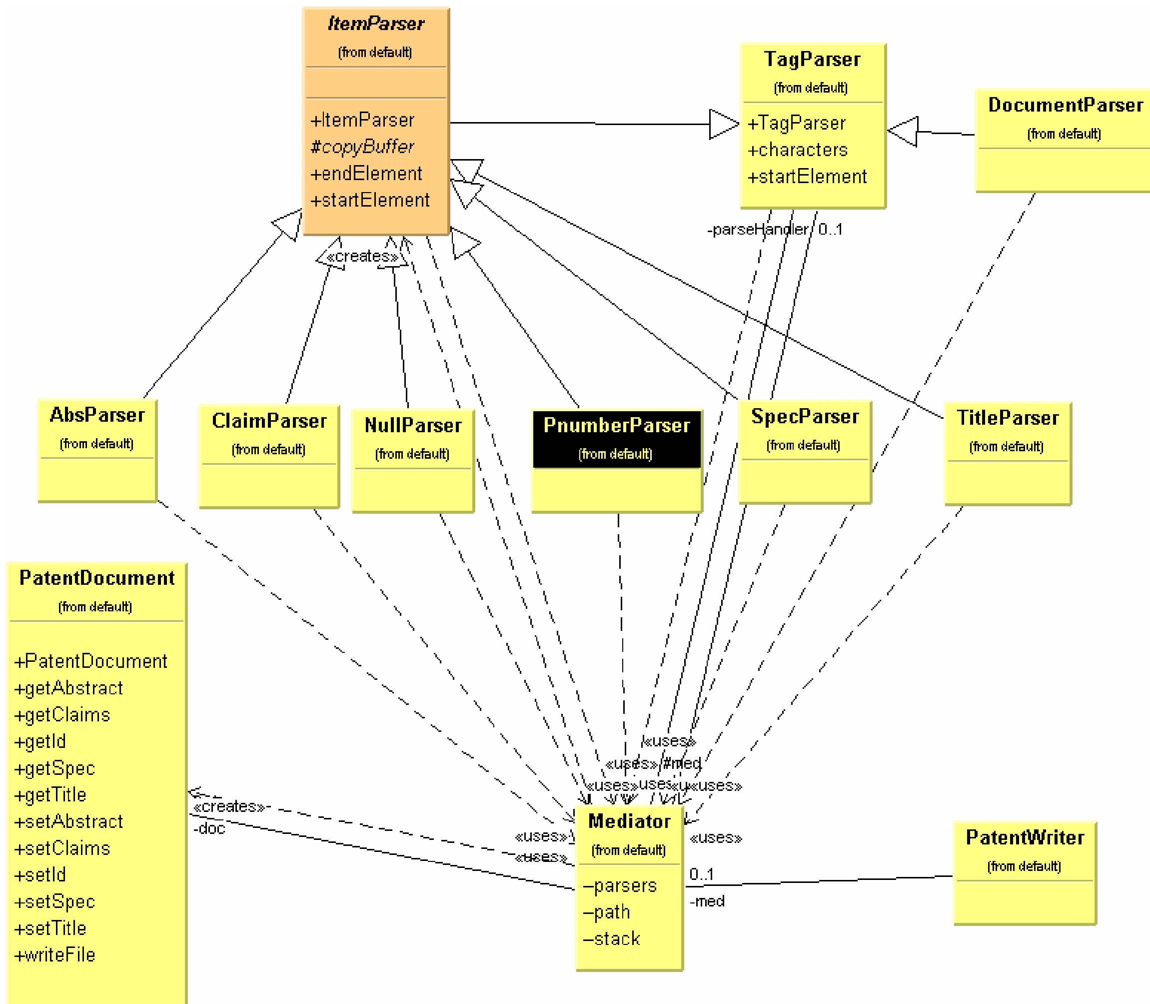

Figure 1 – The class diagram for the nexted XML parser

Leaving the Nest

In summary, we've written a nested XML parser using very little code. The Mediator decides where to send the start, end, and characters method calls, and the parseHandlers decide where to store the characters that have accumulated. There are some more tricks we could learn some time for handling a whole bunch of disparate attributes, and well tackle them in some future flight.