A CHALLENGING TITLE SEARCH

James W. Cooper

Sometimes I marvel at the ingenuity of programmers in making ordinary HTML look so sophisticated on browser screen. For all the limitations of HTML, there are a lot of beautiful web sites that good designers and programmers have constructed without resorting to active components or style sheets.

The downside of all this cleverness is that the HTML itself can be confusing, opaque and downright ugly to read. It sometimes approaches coding “pornography.” (“I don’t know how to define it, but I know it when I see it.”) Some of these pages are generated directly by programmers and designers and others are the result of HTML-generators that take documents in other formats and convert them to HTML.

Imagine, then, the assignment of indexing a bunch of random web pages from a specific crawl, or even on your own private server. It could very well be that you will find several styles, or in the worst case that you will find no two alike.

Despite these difficulties, you could well need to solve the problem of just compiling a list of titles and URLs for the pages on your server. To illustrate the problem of even finding the title of a page, I pulled out 3 HTML documents from different sources to look at. The first is from the IBM Patent Server:

```
<HTML><HEAD>
<META NAME="TITLE" content="Software version management system">
<META NAME="PUBDATE" content="12/10/1985">

<TITLE>Patent Server: 4558413 Detailed View </TITLE>
<META NAME="owner" CONTENT="patserv@almaden.ibm.com">
</HEAD>
```

Here the useful title is in the first META tag. The string inside the <TITLE> tag is much less descriptive.

In a Javadoc from the Java 2 documentation, we find the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<!--NewPage-->
<HTML>
<HEAD>
<!-- Generated by javadoc on Tue Apr 20 23:19:54 PDT 1999 -->
<TITLE>
Java Servlet API Documentation: Class  Cookie
</TITLE>
```

Our title is at least inside the <TITLE> tags here.

However, if you look at one of the examples provided in the JSP Development Kit, also from Sun, you’ll see an even more unexpected approach:

```
<html>
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>

<body bgcolor="#FFFFFF">
```

```
<p><font color="#0000FF"><a href="../servlet/CookieExample">

<h3>Source Code for Session Example<font color="#0000FF"><br>
  </font> </h3>
```

The text inside the <H3> tag is really the only title we have. The text inside the <title> tag is useless here.

So, it might seem impossible to write a title finder for all seasons, and in fact it is pretty difficult. But, if you consider only a constrained set of documents on your own server, you might be able to whittle down the places to find document titles to a tractable few and write code to handle them.

An Abstract Base Class

Let's start by defining a base class which reads the files.

```
/** A base abstract class defining the
behavior of the HTMLDoc class*/

public abstract class HTMLDoc {
     InputFile fl;      //local file variable

 //constructor
 public HTMLDoc(String filename) {
   fl = new InputFile(filename);
 }

 //read next line from file
 public String getNextLine() {
     return fl.readLine();
 }

 //get title in various ways
 public abstract String getTitle();
}
```

We use the InputFile routine we developed earlier and encapsulate it inside a simple HTMLDoc class. Note that we make the getTitle method *abstract* and do not give it a body. Instead, we will derive classes from it for each of the types of HTML documents we've found.

For example, our Patent server documents can be parsed with the following PatentDoc class:

```
//documents from IBM Patent server
 public PatentDoc(String filename) {
   super(filename);
 }

 //In Patent server documents,
 //the title is in the first META tag
 public String getTitle() {
     String s="";
     do {
       s = getNextLine();

     } while (s.indexOf ("<META") < 0  );
     int index = s.indexOf ("content=") + 9;
     s = s.substring (index);
     int last = s.indexOf ("\"");
     s = s.substring (0, last);
     return s;
```

```
 }
}
```

and our JavaDoc documents can be parsed with the JavaDoc class:

```
public class JavaDoc extends HTMLDoc {
   //parse Javadoc documents
    public JavaDoc(String filename) {
        super(filename);
    }
    //In Javadoc files, the title is inside the
    //<title> tag
    public String getTitle() {
        String s= "";
        do  {
            s = getNextLine();
        } while( ! s.trim().equals ("<TITLE>") );
    return getNextLine().trim();
    }
}
```

and so forth. Thus, if we were to write a program to get the titles from these three classes of documents, it might be as simple as

```
public class Titler {
    public Titler() {
        HTMLDoc ht; //instance of base class
    //print out titles from 3 different HTML doc types
        ht = new PatentDoc("4558413.html");
        System.out.println(ht.getTitle());

        ht = new JavaDoc("Cookie.html");
        System.out.println(ht.getTitle());

        ht = new HowTo("sessions.html");
        System.out.println(ht.getTitle());
    }
    //--------------------------------------
    static public void main (String[] argv) {
        new Titler();
    }
}
```

This probably seems pretty straightforward to you, and indeed it really is. However, if you look at the UML diagram in Figure 1 that I generated using JVISION you can see something else pretty clearly.

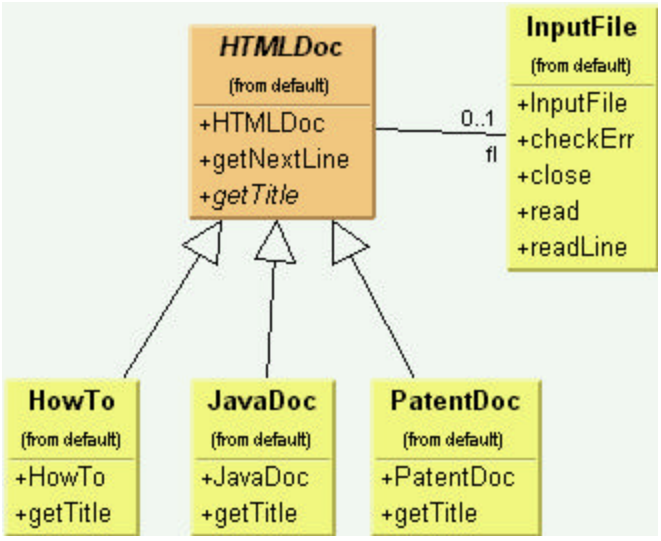

Figure 1 – The UML Diagram of the HTMLDoc class and its derived classes.

The base HTMLDoc class has two methods which are implemented in the base class: the constructor and the getNextLine method. The getTitle method is abstract as shown in the diagram, and it is implemented (differently) in each of the derived classes. This sort of configuration is an example of a Template Method pattern.

Kinds of Methods in a Template Class

As enumerated in *Design Patterns,* the Template Method pattern has four kinds of methods that you can make use of in derived classes:

1. Complete methods that carry out some basic function that all the subclasses will want to use, such as getNextLine in the above example. These are called *Concrete methods.*
2. Methods that are not filled in at all and must be implemented in derived classes. In Java , you declare these as *abstract* methods, and that is how they are referred to in the pattern description.
3. Methods that contain a default implementation of some operations, but which may be overridden in derived classes. These are called *Hook* methods. Of course this is somewhat arbitrary, because in Java you can override any public or protected method in the derived class, but Hook methods are intended to be overridden, while Concrete methods are not.
4. Finally, a Template class may contain methods which themselves call any combination of abstract, hook and concrete methods. These methods are not intended to be overridden, but describe an algorithm without actually implementing its details. *Design Patterns* refers to these as Template methods.

Using Template Methods

Now, the above Titler program parses the three documents and produces the following output:

```
Software version management system
Java Servlet API Documentation: Class  Cookie
```

Source Code for Session Example

Note that the words in the second title are unevenly spaced, because they are unevenly spaced in the source HTML document. We would like to space them more evenly regardless of how we find them, and we can introduce a getCompactTitle method to do this.

```
//get words in title without extra spaces
//This is a template method
 public String getCompactTitle() {
     StringTokenizer tok = new StringTokenizer(getTitle());
     Vector words = new Vector();
     while(tok.hasMoreTokens ()) {
         words.addElement(tok.nextToken ());
     }
     String newTitle ="";
     for(int i=0; i< words.size(); i++)
         newTitle += (String)words.elementAt (i) + " ";
     return newTitle;
 }
```

This method does not need to reside in the derived classes. It can be part of the base class, *even if some of the methods it calls are abstract in the base class.* Thus it seems like the code in the base class is calling the getTitle method in the derived classes. This is referred to as a template method, or what *Design Patterns* refers to as the Hollywood approach, or “Don’t call us, we’ll call you.” Actually, the calls all originate in the derived class and ripple up to the parent class to be resolved.

A Hook Method

Now we don’t *have to* require that the getTitle method be abstract if we don’t want to. We could have a simple method that we usually override, such as one that looks for text inside an <h1> tag.

```
//This is a hook method
 public String getTitle() {
 //in the base class, we'll look for <h1>
     String s = "";
        do {
           s = getNextLine();
        } while (s.indexOf ("<h1>") < 0  );
        s=s.substring (4);
        int i = s.indexOf ("<");
        s=s.substring (0, i).trim();
        return s;
 }
```

If we have a method that we intend to be overridden, that is one we can refer to as a *Hook method.*

Adding a Factory

Of course, it is not likely that we would know from file to file which of the various derived classes to use. Instead, we would probably use a Factory method to compute which the correct class would be. We have written a simple HTDocFactory class below that decides which class to instantiate based on the occurrence of simple terms. This would, of course, need to be expanded for more realistic uses, but illustrates the approach:

```
public class HTMLFactory {
```

```
    String fileName;
    InputFile fl;
//finds strings in an HTML file and creates
//the correct HTMLDoc derived class
    public HTMLFactory(String file_name) {
        fileName = file_name;
        //open file to look for clues
        fl = new InputFile(file_name);
    }
    //-------------------------------------
    //get the correct class
    public HTMLDoc getHTDoc() {
        HTMLDoc ht = null;
        String s = fl.readLine().toLowerCase();

        //scan through file looking for clues
        while( (ht == null) && (s!= null)) {
            if(s.indexOf ("<meta") >= 0) {
                ht = new PatentDoc(fileName);
            }
            if(s.indexOf ("!doctype") >= 0) {
                ht = new JavaDoc(fileName);
            }
            if(s.indexOf ("untitled document") >= 0) {
                ht = new HowTo(fileName);
            }
            s = fl.readLine().toLowerCase();
        }
        fl.close();
        return ht;
    }
}
```

Then, our final main program is one that fetches those class unbeknownst to the programmer and prints out the results:

```
public class FactoryTitler {
    public FactoryTitler() {
        HTMLDoc ht; //instance of base class
    //print out titles from 3 different HTML doc types
    //using a factory to determine which to use
        ht = new HTMLFactory("4558413.html").getHTDoc();
        System.out.println(ht.getCompactTitle());

        ht = new HTMLFactory("Cookie.html").getHTDoc();
        System.out.println(ht.getCompactTitle());

        ht = new HTMLFactory("sessions.html").getHTDoc();
        System.out.println(ht.getCompactTitle());
    }
    //-------------------------------------
    static public void main (String[] argv) {
        new FactoryTitler();
    }
}
```

We see clearly how these classes interact in Figure 2.

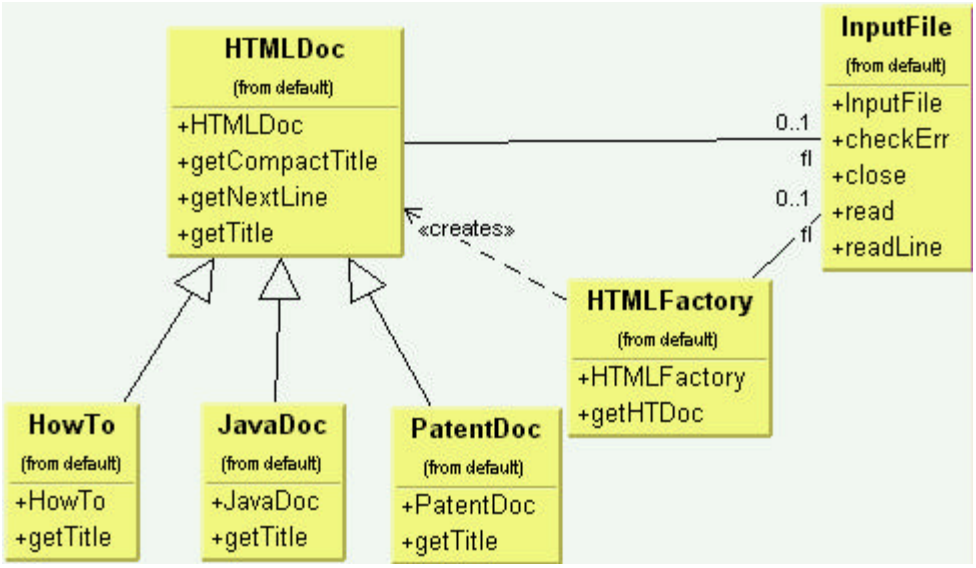

Figure 2 – The UML diagram of the HTML Factory, Hook and Template methods of the Template Method pattern.

Summary

We've seen that the Template Method Pattern describes things you probably have done any number of times. However, by recognizing the various features we see in the Template, we may be able to make more effective use of it in our future programming.

References

1. Gamma, Eric; Helm, Richard; Johnson, Ralph and Vlissides, John, *Design Patterns. Elements of Reusable Software*., Addison-Wesley, Reading, MA, 1995.
2. Cooper, James W. *Java Design Patterns: A Tutorial*, Addison-Wesley, Reading, MA, 2000
3. The IBM Patent Server can be found at http://www.patents.ibm.com