Tag Handling Ideas

# Tag Interpretation Using Reflection

The BrowserMonkey program needs to interpret a large number of tags and do this process in a modular way to allow for future usage of the tag handling system. Most importantly this must be done very efficiently. It would be possible create a html tag handler by using an enormous if-else statement but this would be hard to debug and horrible to reuse or update (for example to new HTML standards). This is why we have decided to use reflection.

## In Theory

The program can use an external file (such as a config file) that is easily editable to lookup required information for use while the program is running. An alternative way of doing this is having the information in the file loaded into the program when is it initially run. This allows for the addition of useful modular features to a program.

Reflection is the process by which a computer program can observe and modify its own structure and behaviour.

Reflection allows the programmer to use multiple different classes and methods in a generic way, perfect for something like html rendering where you have many different processes used to draw the code. Reflection is also very handy for creating a modular piece of software such as a tag interpreter.

Essentially when you instantiate a class using a reflection method you don’t need to know what the class is or what methods it has, the reflection implementation will have methods that will allow you to grab any methods that class has and invoke the one you need.

## In Java

Now I will give a brief introduction of how these techniques could be implemented in Java. Java has a properties class that can be used to implement this kind of idea. There’s a specific convention to use to create a properties file which can be read by included methods. This will then generate a kind of hash table that can be accessed easily.

### External Files

To access the external file to be read into the properties object:

Code:

Properties properties = new Properties();

properties.load(“<root>:\properties.txt”);

You can then access the properties in this file using a Hash table-like key function:

Code:

String thisPropertyIsSetTo = properties.get(thisProperty);

This Code will find the property key ‘thisProperty’ and set the String thisPropertyIsSetTo to the value stored in the properties file as the value for ‘thisProperty’.

### Reflection

Java has a built in class for handling reflection: java.lang.reflect

Here is some code for a simple example of reflection. Code:

import java.lang.reflect.\*;

public class DumpMethods {

public static void main(String args[])

{

try {

Class c = Class.forName(args[0]);

Method m[] = c.getDeclaredMethods();

for (int i = 0; i < m.length; i++)

System.out.println(m[i].toString());

}

catch (Throwable e) {

System.err.println(e);

}

}

}

This code reads the command line arguments and compares the first argument to all available classes in java if it finds a matching class it will output a toString of each method that class contains. If there is no matching class it will throw an error.

## In BrowserMonkey Browser

To apply the above techniques in the BrowserMonkey Browser we will use a Class for each HTML tag that extends an abstract class called TagRenderer that has a render method which for each different class tag will have code used for rendering the item related to the current tag.

There will be an external file that will be loaded into a Map within the Renderer class during its constructor code. We will then use the map to relate the tag the renderer is looking at to the name of the equivalent mini Tag Class. Next the renderer will use reflection to get the necessary rendering method and pass it the current attributes and variables so that it can build the required component based on the tag.

### Pseudo Code

A simple pseudo code representation of how the above ideas would work within the program:

Each mini Tag Class will have the required render method for the tag they relate to. This will be so because it will implement the TagRenderer abstract class. The render method is called repeatedly and relies on information from the former rendering methods that have been run above it in the document node tree.

The signature for the render method is as follows: render(Renderer renderer, TagDocumentNode tag, LayoutRenderNode parent, Map formatting).

The render method will decide what needs to be done with the tag based on the current formatting provided in the formatting variable

In the Renderer:

During Constructor:

Map rendererMap = new Map();

properties.load(<root>:\properties.txt)

tags = properties.propertyNames()

for each tag in tags

String tagClass = properties.get(tag)

TagRenderer thisTagRenderer = new Class.forName(tagClass)

rendererMap.add(tag,thisTagRenderer)

The above code builds the renderMap from the properties file when the renderer is first initialised. This allows the renderer to find the TagRenderers later during the rendering process:

During Rendering Process:

Recursively going over each documentNode:

if documentNode is not plain text

currentTagRenderer = rendererMap.get(Tag)

currentTagRenderer.render(documentNode as TagDocumentNode)

Rendering is a recursive process, document nodes will continue render their children.

## For More Information on Reflection

<http://java.sun.com/developer/technicalArticles/ALT/Reflection/>