

The evolution of DHTML, a non-existent technology.

Craig Knuckles, Joe Hummel
Lake Forest College

Introduction

The First Browsers

The Birth of Browser Objects

The Second-Generation Browser Objects

The DHTML Browsers

The Document Object Model

The HTML DOM -- "Modern Browsers"

What is DHTML?

The Legacy of DHTML

Conclusion

References

Introduction

The original purpose of HTML was to create a logical structure for hypertext documents. The original client software, now maintained by Rose-Hulman Institute of Technology, understood relatively few HTML elements. HTML has changed significantly and the Web has emerged as a platform for distributed computing.

Dynamic HTML (DHTML) has brought Web designers many features sought after in contemporary GUI design for desktop applications; absolute element positioning and a rich event model, for example. That's impressive for a technology which, strictly speaking, does not exist! DHTML is not a markup language, nor is it a programming or scripting language. There is no international standard which defines DHTML.

The term DHTML was coined by marketing executives during the height of the Browser Wars to tout fancy capabilities of their new browsers. With the advent of international standards for HTML, JavaScript (ECMA Script), and Cascading Style Sheets (CSS), DHTML was then loosely defined as the aggregate of these technologies even though no single standard gave it identity or even specified what it should accomplish.

This article begins by exploring how the earliest scriptable browsers, the very same ones which helped the Web to explode, exposed their content to JavaScript as objects. It then ushers in the age of DHTML by examining in detail the first DHTML browsers and their objects. Next comes the standardization of DHTML through the efforts of the World Wide Web Consortium to standardize the Document Object Model (DOM). In particular, this article explores how the browser objects of the first, proprietary DHTML browsers relate to the DOM. Finally, it examines how the general ideas behind DHTML have created a new paradigm for GUI development -- instantiate GUI objects using a markup language and handle events with a programming language.

The First Browsers

The waters surrounding the earliest Web browsers are murky. The book written by Tim Berners-Lee [1] does help to shine some light into those depths. Even the earliest Web browser, simply *WorldWideWeb*, parsed HTML markup code into a tree using the C language. The original code library, *libwww*, is still available today on the Web site [2] maintained by the Wide Web Consortium (W3C).

It is unclear exactly how other of the earliest noteworthy Web browsers (*ViolaWWW*, *Midas*, *Arena*, *Erwise*, *Lynx*) parsed HTML code. Those browsers did help to expand use of the Web somewhat among scientists and programmers, but none of them became big players. The first three listed could load graphics, although not actually embedded in a Web page. The *ViolaWWW* browser was even based upon a contrived programming platform and language. It is extremely unlikely that any of them were scriptable in the sense we think of that today, but *ViolaWWW* may have been the closest.

The Birth of Browser Objects

The Mosaic Web browser pixelated the world's imagination in 1993, followed by Netscape Navigator 1 (NN1) which took over <center> stage in late 1994. But it is in March of 1996 that this tale begins with the release of Netscape Navigator 2 (NN2). It NN2 was the first JavaScript capable browser, and that brings us to the object of our discussion. Now the world had more than a rendition of an HTML file based upon a parse tree. The JavaScript API implemented in NN2 featured a hierarchy of objects which could be manipulated by scripting the Web browser. Such objects would come to be known as *Browser Objects*, and Figure 1 shows those of NN2 -- the mother of all Browser Objects. We have shown a very significant portion of this first Browser Object, opting to forgo a few properties and methods which were never (or rarely) used and to forgo summarizing all of the form element objects such as checkboxes and so forth.



Figure 1 The Browser Objects in the JavaScript API of NN2.

To give an overview of these objects, we have packed quite a lot of information into Figure 1. Let's begin with the window object. The top row lists its properties. For example, `window.location` is a string property which contains the URL of the file currently displayed by the browser window. The second row lists its methods. For example, the method call `window.close()` results in bye-bye window. The third row lists its event handlers. For example, a function can be assigned to `window.onUnload` to cause something to happen when the current page displayed in the browser window is unloaded, like when a new page is loaded. As we explore more Browser Objects, we will adhere to this style of three-row presentation of an object's properties, methods, and event handlers, respectively.

The hierarchy in Figure 1 shows other objects which are nested below the window object. For example, the method call `window.history.back()` is equivalent to hitting the back button on the browser's built-in button bar. As you can see in the diagram, the `history` object has no event handlers. It would be pontificating to continue to go through NN2's Browser Objects one-by-one. But we do offer some examples of the types of dynamic effects made capable by the JavaScript binding in NN2 which continue to be mainstream today.

Figure 2 shows a "rigged link" in NN2. (Yes, ladies and gentlemen, that is really Netscape 2!). The link provides a custom behavior by throwing up an alert window when clicked and by changing the text in the status bar on the `mouseover` event. The original Browser Objects also allowed the Web page itself to be changed after it was fully rendered. Figure 3 shows how a page could be overwritten in NN2. Three seconds after the browser loads it, the original page is completely wiped out and replaced by a new one.

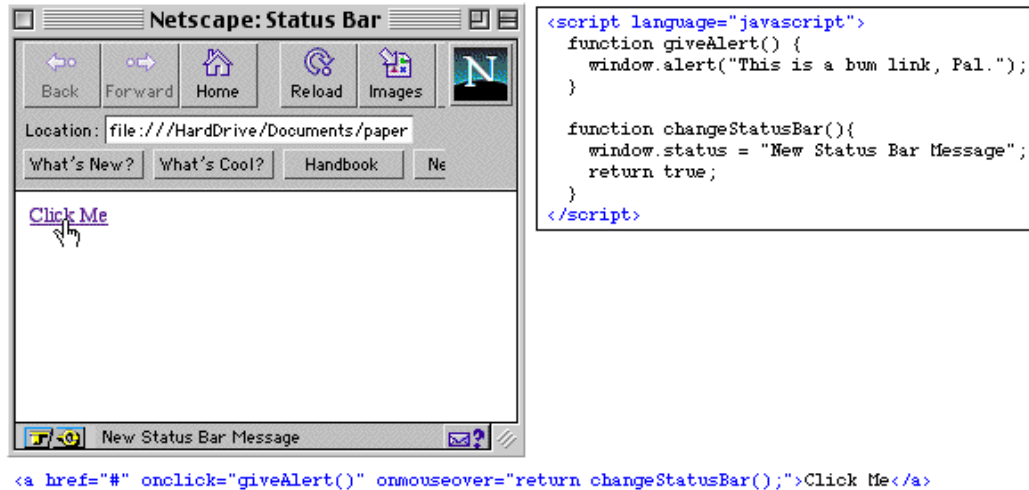


Figure 2 Customized link behavior in NN2.



Figure 3 A page overwrite in NN2.

The most significant addition to the Web's capability added by NN2 was HTML forms. Of course, a form's data could be submitted to a program on the Web server for processing. But NN2 also exposed the form and each of its elements to JavaScript as objects. This enabled the creation of client-side JavaScript utilities such as the rudimentary calculator shown in Figure 4.

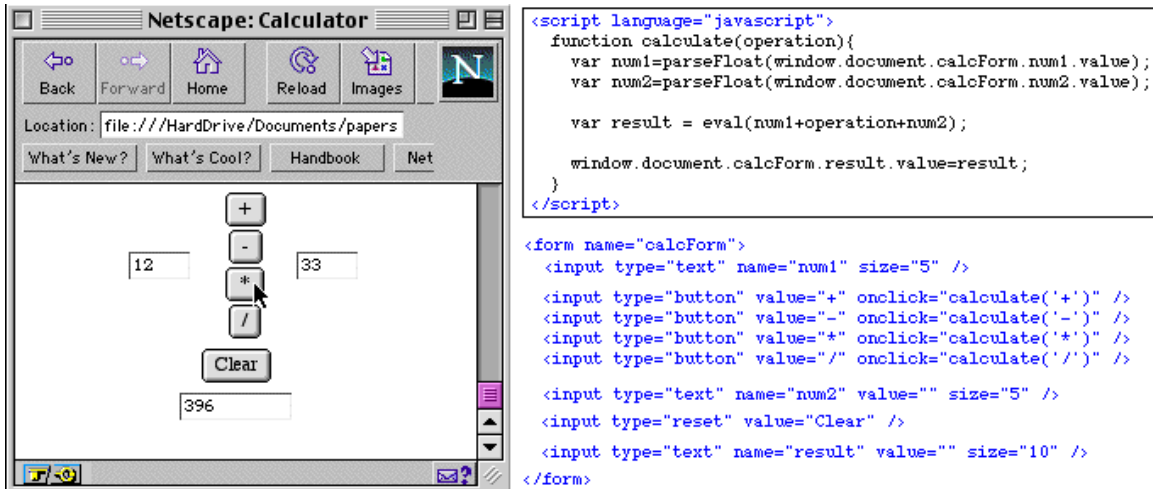


Figure 4 A client-side JavaScript utility in NN2.

Figure 4 is perfect to exemplify the long object references necessary in the original Browser Objects. As the browser reads the HTML code, the Browser Objects are initialized by the markup instructions. In particular, references to the form object, and to the objects which represent the form's elements, are created by the names given as HTML attributes. Thus, protracted object references like

```
window.document.formName.textFieldName.value
```

are used to access the string contained in a text box, for example. In most cases, reference to the parent object, window, can be omitted but we will continue to include it for emphasis.

There are cases where names no longer suffice as object references. For example, in a group of radio buttons that all share the same name (so that only one of them can be selected), the names are rendered useless as object references. Fortunately, the Browser Object provided by NN2 also exposed all of the forms and form elements in a document as a `forms []` and `elements []` array, respectively. The array indexing is determined by the document order of the HTML markup instructions. So for example,

```
window.document.forms[0].elements[7].value
```

supplies alternate object references to access the string in the `result` text box in Figure 4. These built-in arrays of object references use standard indexing beginning with 0. There is only one form in the document, so `forms [0]` references it. The nuance of multiple built-in references got its start in the very first Browser Object.

The Second-Generation Browser Objects

In August of 1996, just a half year after the release of NN2, Netscape Communications released Netscape Navigator 3 (NN3). The interactivity provided by JavaScript enabled Web pages had been a big hit. So the logical course of action for the NN3 developers was to enhance that interactivity. Of course, that meant exposing more aspects of a Web page to JavaScript both by adding more browser objects and expanding the existing ones. Figure 5 shows the expanded capabilities of NN3's Browser Objects.

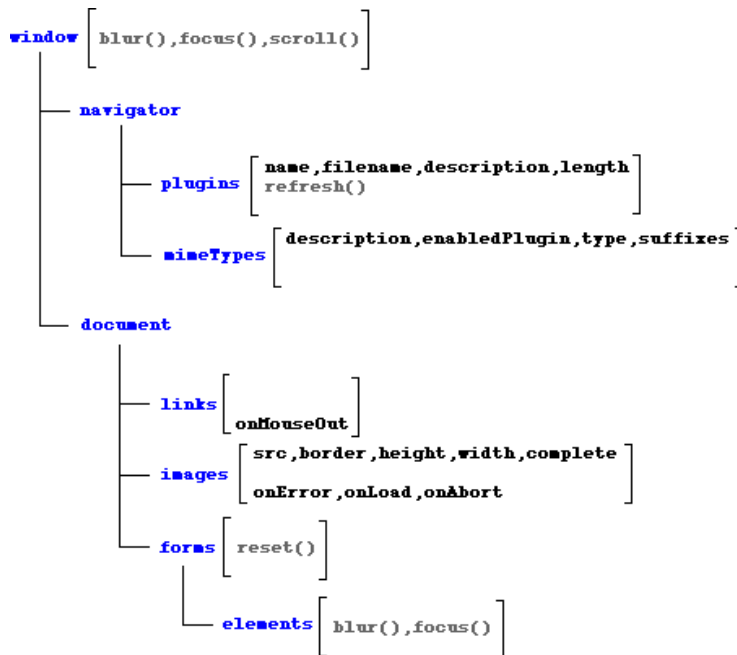


Figure 5 NN3 featured these enhancements in addition to the Browser objects of NN2.

The window object featured added methods which could be used to manipulate the browser's open windows. In particular, a window could now be scripted to be focused (brought to the forefront) and blurred (moved behind other windows). Similarly, form elements could be scripted to be focused (ready for input) or blurred (removed from input focus). NN3 provided the capability to use a proprietary helper application, known as a browser plugin, to aid in delivering new media technologies such as RealAudio over the Web. We won't go into the details, but exposing such media content to JavaScript was the purposes of the new `plugins` and `mimeTypes` objects. It made sense that these objects should be properties of the `navigator` object since it contains the browser-specific information. A Web page with RealAudio content can travel among different browsers, and a short script can determine whether a given browser can handle that content.

The most noteworthy addition was the `image` object, which exposes an embedded graphic in a Web page to JavaScript. By changing the `src` URL of an image with JavaScript after the page had been loaded for some time, the famous rollover graphic effect could be created. As shown in Figure 2, a `link` object could react to the `onMouseOver` event in NN2. But NN3 added `onMouseOut` for links which could be used in conjunction with `onMouseOver` to swap graphics in and out to create a dynamic effect in the page to emphasize that a graphic is in fact active as a link. To the delight of many and the chagrin of others, this technique would become a staple of Web design for many years to come. We overview an image rollover in NN3 in Figure 6 to exemplify the object references.

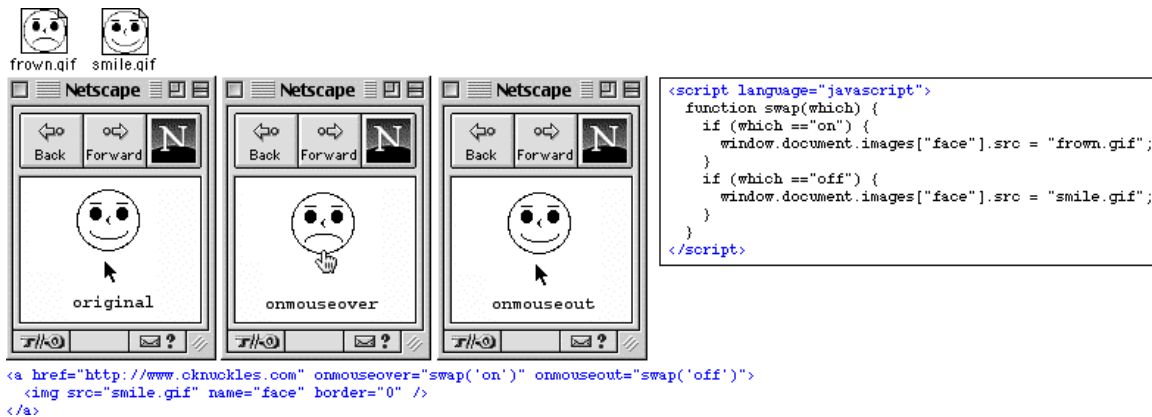


Figure 6 A rollover graphic in NN3.

Microsoft's Internet Explorer 3 (IE3) was also released in August of 1996. Since the client side of the Web was effectively Netscape at that time, the developers of IE3 did no more than to implement NN3's Browser Objects as best they could. They had to or else very little of the current Web would work in their new browser. For that reason the object which holds data about the browser itself in IE3 was named `navigator`. Ironically, it still is to this day and that has to irk at least someone.

There were no standards at this time. Netscape had even added proprietary presentation elements such as `center` and `font` to Berners-Lee's original set of HTML elements, which had been geared more towards logical structure than presentation. Had there been standards upon which to base it, Microsoft might have written their own code engine for their new Explorer browser. Rather, for 2 million dollars, Microsoft purchased rights to browser code from Spyglass, a small company which had developed a browser originating from ideas at NCSA. Marc Andreessen had developed the Mosaic browser at NCSA (National Center for Supercomputing Applications) and other NCSA programmers were privy to its design. Internet Explorer versions 1 and 2 were never really viable on the Web, being little more than beta versions. The IE3 browser was the first legitimate alternative to Netscape. In reality, IE3 could easily have been labeled version 1, but the version 3 designation at least seemed to indicate that this browser was on par with NN3.

The previous section explained how dual object references (names and built-in arrays) existed in the very first Browser objects. In NN3, image objects could be referenced in three different ways. Assuming the image is named by an HTML attribute, the following three references are equivalent. (The last one assumes the image is the first in the document, hence it has a 0 index in the built-in `images []` array.)

```

window.document.imageName.src
window.document.images["imageName"].src
window.document.images[0].src

```

This would cause confusion to casual Web programmers for many years to come, and likely still does. This would later influence the W3C's definition of an `HTMLcollection` type of node. In the first one, the name can be used for easy reference to the object. The second one facilitates an object reference when its name is stored in a variable as a string. Of course, referencing by arrays is useful for iteration over groups of objects.

The DHTML Browsers

Netscape Navigator 4 (NN4) was released in June of 1997 and Internet Explorer 4 (IE4) was released 4 months later in October of 1997. We first take a brief look at the similarities between these two browsers. First, both browsers made a stab at implementing the fledgling Cascading Style Sheets (CSS) recommendation. Many proprietary HTML elements, mostly geared toward presentation rather than content description, were being introduced by the browser vendors. To combat that, the World Wide Web Consortium (W3C) standardized CSS1 in 1996, just in time to influence the version 4 browsers.

There was some commonality between the Browser Objects of the version 4 browsers. The significant additions shared by NN4 and IE4 are shown in Figure 7. One addition was that HTML form elements could react to additional mouse and keyboard events. More significantly, the browser windows themselves could be scripted to move, resize, and scroll. Moreover in the version 4 browsers `setInterval` could call a function repetitively based upon a time delay. Recall that in the very first Browser Object, the `setTimeout` method could call a function once after a preset time delay.

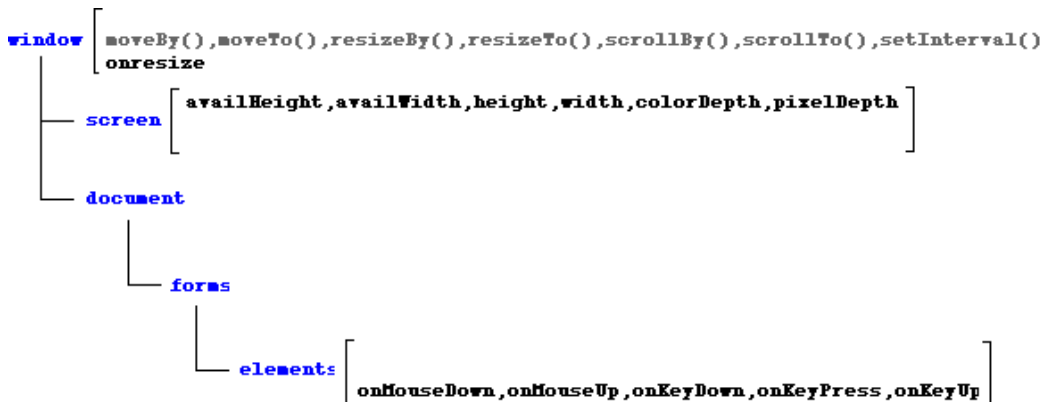


Figure 7 The version 4 browsers had this added functionality in common.

For example, consider the following line of JavaScript code.

```
window.setInterval("window.moveBy(10,10)",500);
```

This code will cause the window to move down and to the right by 10 pixels in each direction every half second. That will actually make the window animated as it moves toward the lower right corner of the screen. (The same effect could be created by setting up a function which calls itself recursively, but then the "speed" of the animation would be determined by how fast a given computer can process the function calls and redraw the window.) The read-only properties of the `screen` object you see in Figure 7 could be used to fit windows to screens, keep a window from moving off the screen, and the like.

But the tale of these browsers gets interesting when you look at their differences. Both browser vendors were trying to make a push both to garner more market share (or hang on to it) and to push the envelope of interactivity on the Web. It is not clear exactly whose mouth first blurted out the term Dynamic HTML (DHTML), but the general consensus is that the propagation of that term is the direct result of marketing campaigns by these two main browser vendors.

We will later discuss how DHTML would come to be defined in retrospect. But the best way to understand the origins of DHTML is to contrast the first two DHTML browsers. We won't contrast every nuance, of course, but will stick to the additions to the Browser Objects geared toward adding dynamic functionality (or novelty, depending upon your school of thought) to the Web.

We begin with the enhancements unique to NN4. On the HTML side, NN4 introduced a proprietary layer element.

```
<layer id="x"> block1 </layer>
<layer id="y"> block2 </layer>
<layer id="z"> block3 </layer>
```



According to document order, each layer is given a higher `z-index` which defines its "stack order" on the page. But each layer must be set to `position: absolute` using a CSS rule to create the stacking effect. Moreover, in the graphic on the right, their absolute positions (coordinates) are set using CSS so that the layering is obvious.

Stacking elements was not overly useful unless behaviors could be supplied. As pictured in Figure 8, NN4 exposed each layer to JavaScript as an object. For example, reference to the first layer object defined

just above is obtained by using its unique `id` value. Depending on your needs in a given script, one of the two references shown below could be used.

```
window.document.x
window.document.layers["x"]
```

Using such a reference to a layer, some 22 properties of the layer could be accessed in a script. And of course, the `document` object contained within a given layer has many properties itself.



Figure 8 NN4 added layer objects to enable dynamic content.

The dynamic effects which could be accomplished by manipulating the layers with JavaScript included changing the `z-index` (layer number), changing a layer's `visibility` (show or hide), and changing the `top` and `left` properties (its coordinates). Thus, animations are possible where layers move across the screen, toggle between hidden and visible, and even "re-stack" themselves.

But NN4's layering scheme was cumbersome to use since each layer was considered to contain its own `document`. So when layers are nested

```
<layer id="x"> <layer id="y"> <layer id="z"> </layer> </layer> </layer>
```

the object references become very long. For example, either of the following references would access the `visibility` property of the inner layer above.

```
window.document.x.document.y.document.z.visibility
window.document.layers["x"].document.layers["y"].document.layers["z"].visibility
```

This cumbersome object model was one drawback of NN4. But in retrospect, its main limitations were in its poor CSS support. First, its CSS was very buggy. Lists of NN4 CSS bugs are pages long. But worse yet was the fact the NN4s layer object model failed to expose CSS properties to JavaScript. Thus, you could not write scripts to dynamically manipulate an object's CSS presentation.

With respect to layering, the IE4 browser took a similar approach although with a different implementation. Layers were created in IE4 using the HTML `div` element.

```
<div id="x"> block1 </div>
<div id="y"> block2 </div>
<div id="z"> block3 </div>
```



But again, to achieve the layering seen in the graphic to the right, `position: absolute` would need to be set using a CSS rule and also the block sizes, positions, and so forth.

The object references are in stark contrast to NN4, however. IE4 included a built-in `all` object to provide direct object reference to every HTML element in the page. For example, either of the object references below point to the first `div` block defined above.

```
window.document.all["x"]
window.document.all.x
```

The `div` block `x` could be nested deep inside the HTML code (inside other `div` blocks or otherwise) and the `all` object still finds it. The `all` object crawls the tree for you.

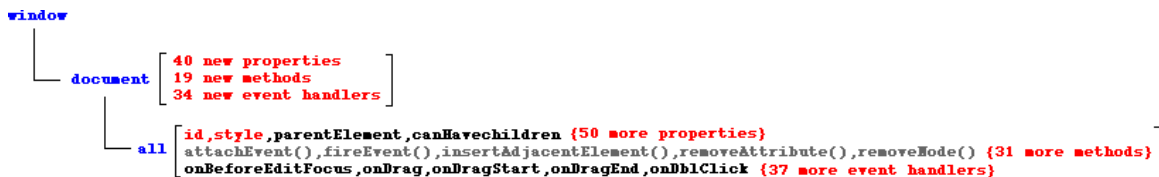


Figure 9 IE4 added the `all` object and radically expanded the objects in general.

Microsoft tried to make a statement with the release of IE4 by radically expanding the Browser Objects. Figure 9 gives an overview. As you can see, the objects are far too extensive to summarize. But we have indicated how massive they are. Among the `all` properties and methods we have chosen to list are ones related to manipulation of the tree model of the parsed HTML file. This enabled the Web page to be "reflowed" in IE4 as shown in Figure 10. When the sports link is clicked, the other two blocks move down enough to allow the other two blocks to be displayed.

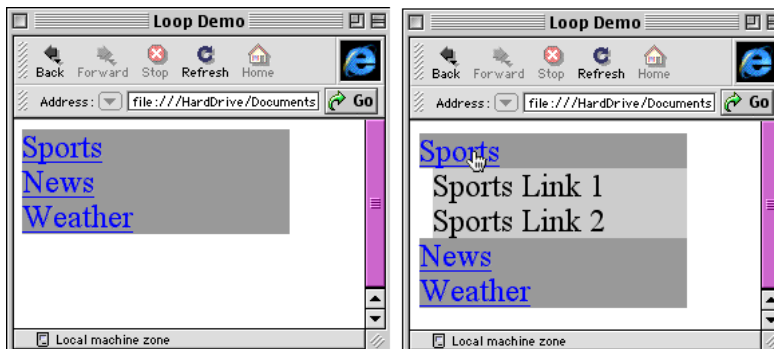


Figure 10 The IE4 Browser Objects allowed a script to change the page layout flow.

This is not done with layers but by actually changing the "planar" page layout flow after the page is fully rendered. The underlying principle is to be able to add new element nodes (sports link blocks, in this case) to the parse tree for the page from within a script. This capability, not present in NN4, was a significant advancement in scripting capability in Web browsers.

IE4 also could do most of the other noteworthy dynamic stuff that NN4 could: moving layers, showing/hiding layers, re-stacking layers, etc. One distinguishing feature of IE4 was the exposure of `all` HTML elements to JavaScript via its unique `id`. But the CSS support of IE4 and, in particular, its exposure of an element's `style` property is its main claim to fame. The `style` object as exposed to JavaScript has over 150 properties, most of which can be changed by a script. This gave JavaScript unprecedented control over most any aspect of a Web page.

The Document Object Model

Viewing an electronic document's structure as a tree goes back to the Standard Generalized Markup Language (SGML), and perhaps before. Given proper nesting of markup instructions, an electronic document can be read into memory as a data structure -- a tree (or forest).

But the Browser Object model which grew out of the Web viewed an HTML document as a tree with behaviors. Thus we can parse an HTML file into an object tree and then manipulate it programmatically. The structure and interface which allows for this has since been called the Document Object Model (DOM). In 1998, the W3C released recommendations for both the eXtensible Markup Language (XML) and DOM Level 1 (DOM1). The DOM1 specification [3] asserts that the DOM provides:

- the interfaces and objects used to represent and manipulate a document
- the semantics of these interfaces and objects - including both behavior and attributes
- the relationships and collaborations among these interfaces and objects

In short, the DOM defines a language independent API which allows well-formed XML documents to be manipulated by a programming language.

Exploring the DOM in detail is well beyond our objectives here, but a brief look at it is instructive for this article. Figure 11 shows the DOM tree structure for a very simple XML file. The diagram is borrowed from the book [4].

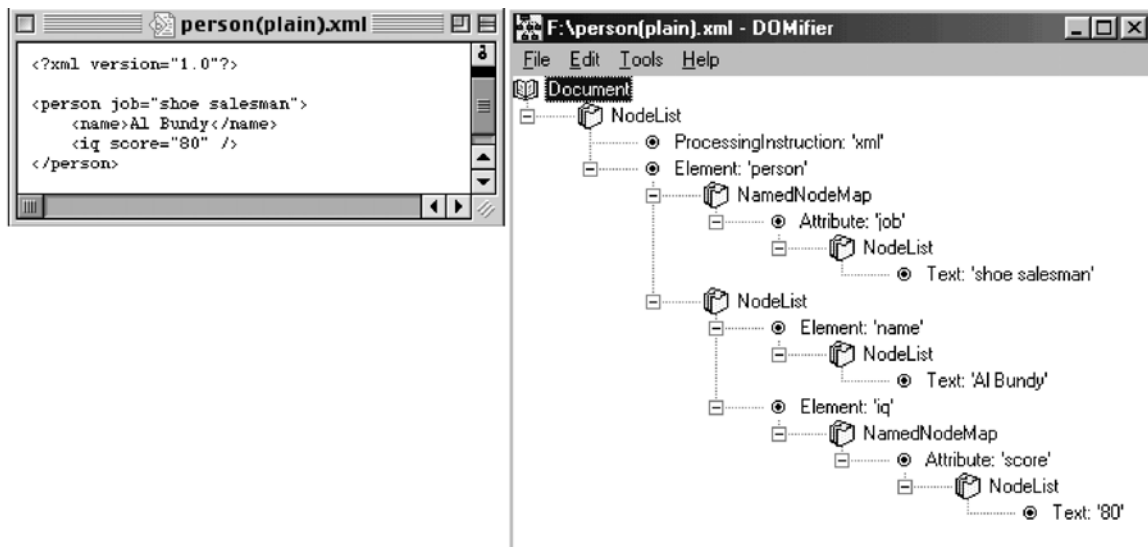


Figure 11 The DOM tree for an XML file.

If the DOM were a place on Earth, it would be called Node City. Everything (elements, attributes, attribute values, element content) is a node and there are NodeLists and/or NamedNodeMaps at each level.

Below each element is a `nodeList` node called `childNodes` which serves in an array-like capacity to index all of an element's children.

```
root.childNodes.item(0).childNodes.item(0).nodeValue; //Al Bundy
```

Below each element is also a `namedNodeMap` node which serves in a hash-like capacity to provide access to its attributes by name.

```
root.attributes.getNamedItem("job").nodeValue; //shoe salesman
```

The point to take from this is given a language binding, JavaScript in this case, you can manually "crawl" the tree to access any piece of data in the file.

Fortunately, the DOM also provides methods which crawl the tree for you. DOM1 provides `getElementsByTagName()` method, which returns a `namedNodeMap` of all elements with a specific name, regardless of their place in the tree. The DOM2 recommendation added the `getElementById()` method, which returns the `Element` node matching a given `id` attribute value, regardless of its place in the tree. This is reminiscent of IE4's `all` object.

Conversely, you can add more data to the DOM at any point using methods such as `createElement()`, `createAttribute()`, `createTextNode()`, just to name a few.

The HTML DOM -- "Modern Browsers"

Now let's pop back to 1997 and the DHTML browsers IE4 and NN4. In an effort to keep the Web from becoming quagmired into a proprietary system, it was imperative that the W3C standardize the JavaScript API for HTML documents. It was imminent that Microsoft's IE was going to win the browser wars since they were bundling it with their dominant desktop operating system.

Two things were clear. One, the original Browser Objects were too well entrenched to go away. Two, it would be desirable for virtually all aspects of an HTML document to be exposed to JavaScript in a manner similar to the XML DOM, which was under development at that time. The end result would be a DOM which could live in both worlds -- the HTML DOM.

The DOM1 recommendation contains two parts: Core DOM, which pertains to XML documents in general and HTML DOM which is geared toward HTML documents in Web browsers. We took a brief look at the Core DOM in Figure 11 and the surrounding discussion. When an HTML document is well-formed as an XML document (i.e. it's an XHTML document) it can be represented in memory as a Core DOM Object (with all the rights and privileges appertaining thereunto).

Without defining what it looks like whatsoever, the HTML DOM defines DOM Level 0 (DOM0) as the Browser Objects supported by the version 3 browsers. For the most part that means the NN3 Browser Objects -- the "sum" of those shown in Figures 1 and 5. This ensures that when new browsers are built according to HTML DOM standards, the billions of "old" Web pages will still work properly. That is, HTML DOM is backwards compatible with huge chunks of the Web. Figure 12 depicts DOM0 as part of the HTML DOM.

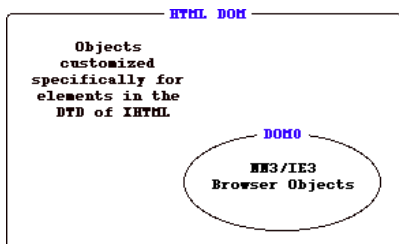


Figure 12 An abstract look that the HTML DOM.

The rest of the HTML DOM makes good use of the fact that XHTML has a Document Type Definition (DTD) which defines exactly which elements and attributes the language supports, their allowed containments, and so forth. Thus, HTML DOM can make use of the functionality of the Core DOM but in a way to utilize the specific functionalities of the standardized XHTML elements. A more concrete overview of the HTML DOM is provided in Figure 13.

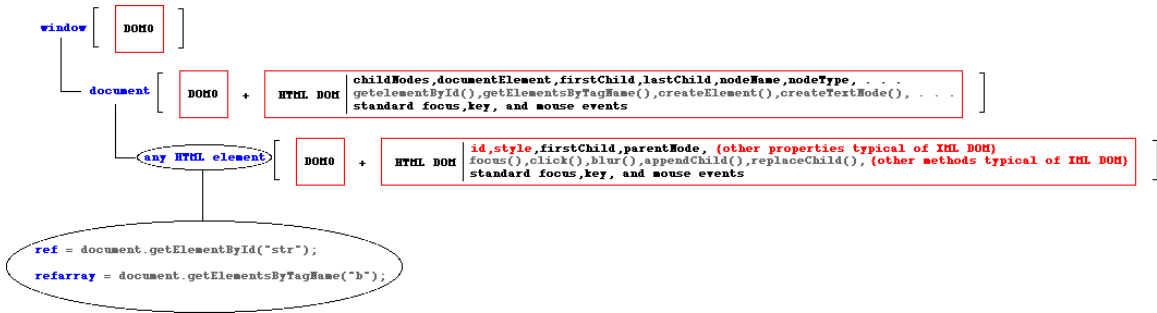


Figure 13 An overview of the HTML DOM.

In practice, things like manipulation of browser windows, image rollovers, and client-side processing of form data are often still done using DOM0. The window is exposed to JavaScript entirely in DOM0 fashion. The HTML DOM's document object is over half DOM0. For example, of the document object's 38 properties, 20 come from DOM0, 6 come from IE4's DHTML DOM, and the remaining 12 are W3C additions. However, the object representing any HTML element is mostly a W3C creation.

DHTML effects such as manipulating layers and style properties need the functionality enabled by the full HTML DOM. Individual elements are given an `id` attribute with a unique value. References to the object representations of those elements can be obtained by using full CORE DOM type references, somewhat similar to those shown following Figure 11. But more often, the tree crawling methods shown at the bottom of Figure 13 are used to pluck object references to elements (or arrays of elements) out of the tree, irrespective of how they are nested. The exposure of any HTML element (and its style properties) to JavaScript in the HTML DOM is quite IE4-like. IE4 was the more successful DHTML browser in that many of its features on (usually with different names) in the HTML DOM.

The HTML DOM brings somewhat more power than the DHTML browsers (IE4 and NN4) in terms of directly manipulating nodes in the tree. But more importantly, HTML DOM provides an international standard so that DHTML can be implemented uniformly in all "modern browsers." And for the most part that means that object references to elements can be obtained without browser sniffing and workarounds to accommodate things like NN4's *layer DOM* and IE4's *all DOM*.

People who script Web browsers for a living have invented their own nomenclature to describe all of the different Browser Objects and DOMS. Of course, the NN3/IE3 legacy Browser Objects are uniformly called DOM0. An excellent cross-browser JavaScript resource [5] defines both NN4's layer DOM and IE4's all DOM as the *intermediate DOMS*. Our terminology, the *original DHTML DOMS* seems equally apt. The site [5] defines *the advanced DOMS* as being either of the intermediate DOMS and the HTML DOM.

We would prefer to say that *the advanced DOM* for Web pages is W3C's HTML DOM. Then, an *advanced browser* (or *modern browser*) would be one that implements the advanced DOM. Using that definition, the most common modern browsers are IE5+, Opera5+, any Mozilla browser (NN7+, Firefox), and Apple's Safari (which is based on the Linux Konqueror browser). Of course, we would be remiss not to mention W3C's Amaya browser/editor.

So the tale of the second generation Browser Objects of NN3 and IE3 has a happy ending; they live on as DOM0. They worked very well for programmers then for what they were meant to accomplish and they still work well for those purposes today. But the tale of the DHTML browsers is a sordid one. Some would recount that entwined tale in terms of the browser wars and the political battles over what W3C's HTML DOM should look like. But a client-side Web programmer would simply offer the following concise tale, and then let the issue rest in peace.

```

if (document.getElementById) {
    ref = document.getElementById("someId");
}
else if (document.all) {
    ref = document.all["someId"];
}
else if (document.layers) {
    ref = document.layers["someId"];
}

```

```

}
else
  ref = null; // non DHTML browser
}

```

Many commercial-grade Web programmers still script for three DOMS to maximize their audience, but that need will abate as the version 4 browsers and their pioneering DHTML DOMs ride off into the sunset.

What is DHTML?

It is telling that Tim Berners-Lee makes no mention whatsoever of "Dynamic HTML" or "DHTML" in his own account [1] of the creation and history of the Web. Indeed, fancy user interfaces are a by-product of the Web, not one of its underpinnings. W3C offers the following definition. "DHTML is the marketing term applied to a mixture of standards including HTML, style sheets, the Document Object Model [DOM1] and scripting. However, there is no W3C specification that formally defines DHTML." [6]

The title of this paper facetiously asserts that DHTML doesn't exist. Since it obviously does exist, it is more appropriate to ask whether it is a technology. Is a technology defined as a set of rules or standards which uniformly cements its existence? Merriam Webster's Online Dictionary (Merriam-Webster, 2004) defines technology as "the practical application of knowledge" or "a manner of accomplishing a task." We conclude that a technology is the working result of some novel way we figure out how to do something. Thus, DHTML may not be well-defined, but it is a technology.

Some definitions are much more restrictive. For example, many people (and books) simply characterize DHTML as the changing of style sheets or properties using JavaScript. However, we feel that the style-only definition is overly restrictive. Figure 14 illustrates the crux of the matter. If you manipulate the DOM in one way, say a DOM0 type window movement or HTML form calculation, is that not DHTML? To create a true DHTML effect do we need to formally access a style property, or does changing the page layout flow by inserting a new element node and its content using `createElement()` and `createTextNode()` qualify? We either have to split hairs or concede that, if you can script the HTML DOM to make something change, it's DHTML.

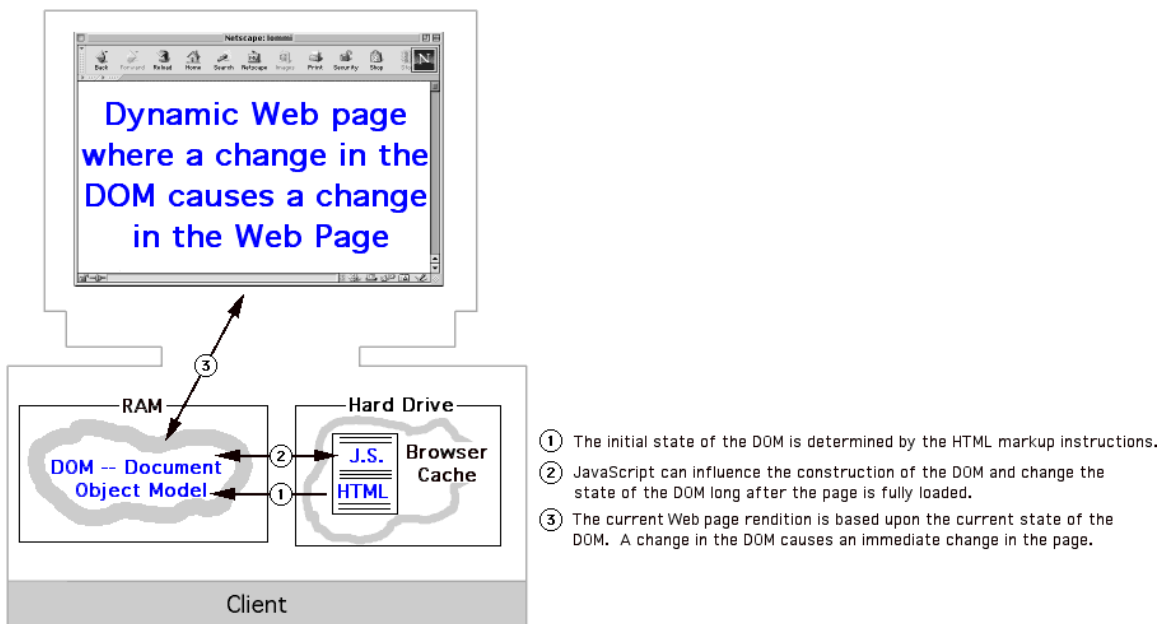


Figure 14 DHTML -- manipulating the DOM using JavaScript.

Or perhaps the term DHTML should just be scrapped in favor of "browser scripting," or something like that. Browser scripting has evolved from simply coding client-side utilities using HTML forms or providing some fancy effect to entice the surfer to come back as opposed to visiting some other non-dynamic site. With modern browsers and a standardized DOM, Browser scripting has now enabled Web pages to contain interactive elements (sliding menus and bars, popup text for tips or hints, collapsible/expandable navigation bars, etc.) more representative of what would be considered a contemporary Graphical User Interface) GUI.

The Legacy of DHTML

With a standardized DOM, which offers plenty of Browser Objects and exposes over 150 style properties of HTML elements, the practice of creating fancy, intuitive GUIs on the front end of Web applications is just starting to hit full stride. This does enhance the user's experience and overall functionality of the site, but it is arguable as to whether this is a fundamental step forward for the Web. It's cool, but it doesn't enable Web applications accomplish that which they otherwise wouldn't, per se.

Perhaps the most important legacy of DHTML is that it has introduced a new paradigm in GUI development -- create a GUI using a markup language, and script the events using some programming language. The deployment paradigm is not significantly shifted given that JAVA has facilitated easily portable GUI applications (applets) since the mid 1990s. But in the DHML GUI paradigm, the data is passed around the Internet in ASCII format and parsed by a markup engine. That takes full advantage of the Web's built-in text-based transport mechanism.

New GUI markup languages such as Mozilla's eXtensible User-interface Language (XUL), pronounced "zool" [6], and Microsoft's eXtensible Application Markup Language (XAML), pronounced "zammel" [7], are modeled on this new paradigm. Both use an XML-based markup language to create the GUI and some external programming or scripting language to react to user events. We proceed with simple examples first from XUL and then from XAML.

The following XUL document contains markup code which creates a window with a checkbox, text box, and a command button.

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>

<window
  title="XUL Demo"
  orient = "horizontal"0
  xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

  <script src="event.js"/>

  < vbox style="height:100px">
    <checkbox id="the-checkbox" checked="true" label="An option."/>
    <textbox id="the-textbox" value="Initial text"/>
    <button label="Click Me!" onclick="respond()"/>
  </vbox>

</window>
```

All of the markup elements are qualified (by a rather odd URI) into XUL's namespace. You can see minor similarities between this markup language and HTML, although those similarities don't go much further. XUL provides for much more sophisticated form elements such as sliding range bars, trees where the branches are collapsible/expandable, and even some built-in data validation controls.

To demonstrate adding functionality to this window, we simply program the button click to change the "Initial text" in the text box only if the checkbox is selected, which is its default as you can see in the markup code. A button click calls a JavaScript function named `respond`, which resides in a separate file which we chose to name `event.js`. That function is shown below.

```
function respond () {
```

```

    if (document.getElementById('the-checkbox').checked) {
        document.getElementById('the-textbox').value = "Button was Clicked."
    }
}

```

If you are familiar with W3C's DOM, then this code should be no mystery. XUL markup is exposed to JavaScript using that DOM.

Such a XUL window can be used in two ways. First, the XUL file shown above can be loaded directly into a XUL-capable browser, such as Netscape 7 or another Mozilla based browser. In that case, the XUL file functions much like a traditional, scripted Web page. But a big difference is that XUL can create much more advanced GUIs.

Another way is to use it as a stand-alone application. All of the relevant files (XUL markup, script file, installer script file) can be packaged into a single-file installer and very easily transported and installed on any platform. In fact, the Mozilla-based Web browsers are themselves XUL applications with the GUIs created by markup code. It is thus quite easy to customize the look of the browser (give it a new "skin") simply by changing the style sheets applied to the markup code.

XAML can also be used in two ways. First it can be loaded directly into a Web browser as would the XAML file shown below.

```

<?xml version="1.0"?>

<Window
  xmlns="http://schemas.microsoft.com/2003/xaml"
  Visible="true">

  <SimpleText Foreground="Black" FontSize="18">Hello World!</SimpleText>

</Window>

```

All the markup elements are qualified into Microsoft's namespace for XAML. Like HTML, XAML can also be scripted to create dynamic behaviors such as mouse-overs and animations. But XAML offers much more sophisticated GUI elements. The next major release of Windows --- code-named "Longhorn" --- is set to introduce XAML. Moreover, their Longhorn browser, presumably the next version of Internet Explorer, will be fully XAML capable.

The second way to use XAML is in conjunction with code written in a traditional programming language such as C#, VB, or C++ to create an application. The markup is used to describe the GUI, the code is used for event handling and non-UI tasks, and the two are compiled together to form the application. Below we show the XAML file which contains the markup to create a GUI with a check box, text box, and command button.

```

<?xml version="1.0"?>

<Application
  xmlns="http://schemas.microsoft.com/2003/xaml"
  xmlns:def="Definition"
  def:Language="C#"
  def:Class="ExampleApp:CodeBehindClass"
  def:CodeBehind="CodeBehindClass.xaml.cs">

  <FlowPanel>
    <CheckBox ID="chkBox" Checked="true">An option.</CheckBox>
    <Text ID="txtBox">Initial text</Text>
    <Button Click="OnClick">Click Me!</Button>
  </FlowPanel>

</Application>

```

The C# code, which resides in a separate file, to handle the click event is shown below.

```

namespace ExampleApp
{

```

```

public partial class CodeBehindClass
{
    private void OnClick(object sender, ClickEventArgs e)
    {
        if (chkBox.Checked)
            txtBox.TextRange.Text = "Button was Clicked.";
    }
} //class
} //namespace

```

Note here that the objects created by the markup are interfaced in a way vaguely similar to that of DOM0, but that the object and property names are somewhat different. In general, XAML object model is quite different from W3C's DOM.

One of the goals of XAML is to allow the description of GUIs so applications can run in a variety of .NET environments: Web, mobile, and traditional desktop. Indeed, the goal of XUL is similar (sans .NET). Since these markup languages are well-formed XML, they can easily be combined with other XML based technologies such as XHTML and Scalable Vector Graphics (SVG), or a proprietary version thereof, to create powerful, portable GUI applications. These markup-created GUIs are even ideal to provide portable controls for XML-based Web services.

Perhaps the most important feature of this new paradigm is that the GUI's data can be separated from the programming code which processes it. In these simple examples it was not apparent that the data can be separated from the interface definition. However, when used in conjunction with other XML-based languages, such as Resource Description Framework (RDF), the data can be entirely separated from the user interface. XML data rigorously defined by RDF templates, not encumbered with presentation details whatsoever and kept in totally separate files, can be bound to XUL GUIs to create complex GUI elements on the fly and styled to display on a variety of traditional and mobile devices. Separate XML data sources can also be used as "data bindings" for XAML GUIs.

When proprietary Web technologies look promising and practical, the W3C is usually not far behind. The Xforms specification defines yet another an XML-based markup language which can be used to create advanced GUIs. Being XML-based, Xforms can be used to enhance XHTML pages, or other XML-based languages such as SVG.

Conclusion

The first scriptable Browser Objects were provided by early Netscape Browsers. As scripting on the Web became more popular, Microsoft and Netscape introduced proprietary additions in their Version 4 browsers both to HTML and to how rendered HTML documents were exposed for scripting. In fact all significant additions to the original HTML markup language and its functionality in Web browsers were proprietary.

Fortunately, no corporate entity came to control the Web, and the W3C standardized HTML and its programmatic interface in Web browsers -- the DOM. Although the DOM is geared toward general XML documents, the HTML DOM is specialized for Web pages. In particular, the HTML DOM is backwards compatible with Netscape's original browser objects. That portion of the HTML DOM is called DOM0, and useful for handling tasks such as window control, image rollovers, and client-side processing of HTML forms.

The full HTML DOM can be more specific than the general XML DOM since can be tailored specifically for the fixed DTD of HTML. It is this object model which enables most of the dynamic effects in Web pages we call DHTML. The full DOM resembles IE4's DHTML DOM much more so than that of NN4, whose layer DOM was pretty much a lost cause. As the older browsers fade away, the DHTML effects we can use in practice are getting more elaborate since all of the HTML DOM can be exploited.

Even though DHTML was originally not part of the intended functionality of the Web, its evolution has brought about a new paradigm in GUI Development. New XML-based markup languages such as XUL and XAML describe components a GUI, leaving its behaviors to be provided separately by a scripting or programming language. Such a GUI can be portable on the Web or installed as a traditional desktop application. Using the full strength of this new paradigm, a GUI can be styled (using CSS, XSLT, or otherwise) differently to render appropriately in Web browsers and various portable devices. Moreover, data sources can be kept completely separate from the markup instructions for the GUI and the programming which gives it functionality.

References

- [1] Berners-Lee, T., *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor*. San Francisco, Harper, 1999.
- [2] <http://www.w3c.org>, Accessed Sept 12, 2004.
- [3] <http://www.w3.org/TR/REC-DOM-Level-1>, Accessed Sept 12, 2004.
- [4] Knuckles, C., Yuen, D., *Web Applications: Concepts and Real World Design*. Now York, John Wiley & Sons, 2005.
- [5] <http://www.quirksmode.org>, Accessed Sept 12, 2004.
- [6] W3C Glossary, <http://www.w3.org/2003/glossary/subglossary/wcag10.rdf/>, Accessed Sept 12, 2004.
- [7] <http://www.mozilla.org/projects/xul/>, Accessed Sept 12, 2004.
- [8] Rector, B., *Introducing Longhorn for Developers*, Microsoft Press, 2004.