Jan Steinman Barbara Yates

Documents on the Web

ONE YEAR AGO, we launched this column with an essay on project documentation. We followed up in September 1995 with some code sketches for implementing a hyperliterate documenting system for VisualWorks under Envy.

Now the World Wide Web has taken the planet by storm. This is both a crisis and an opportunity. Those who ignore the Web risk ending up as roadkill on the infobahn. On the other hand, those who approach the Web with a thoughtless, knee-jerk reaction are just as likely to fail.

The latter approach can significantly reduce productivity. "MegaCorp's goal is to have all project documentation available on the Web," sounds nice, until it is followed with "therefore, all programmers will immediately attend HTML training, so they can create their documentation for the Web." This makes it very difficult to fulfill our first principle of hyperliterate programming:

The documentation for a thing must be on the same conceptual level as that thing.

The developers will have to "shift mental gears" to get out of Smalltalk mode and into HTML mode. Knuth's initial concept of literate programming required programmers to learn and use a complex textual mark-up language, which muddled their conceptual space. Eventually, tools for LaTeX emerged, but literate programming never quite reached the masses, largely because of the cognitive dissonance between coding programs and coding documentation.

"No problem, we'll invest in Web authoring tools to make the HTML part easy." This also kills productivity, which is now a victim of ignoring our second principle of hyperliterate programming:

The documentation for a thing must constantly and accurately describe that thing.

If developers are leaving Smalltalk to write Web pages, no matter how good their Web authoring tools, neither their code nor their documentation is going to be as good as it would be if they combined the two activities in one environment. One or the other will suffer, and it is almost always the documentation that is not "constantly and accurately" describing the code.

In reality, the only principle of hyperliterate programming that Web authoring partially fulfills is the third one:

The documentation for a thing must be accessible; by creators, their peers, reusers, reviewers, end-user documenters, and the merely curious.

But wait, who does Web authoring give improved access to? Certainly not the creator or their peers, who must now have both Smalltalk and a Web browser running in order to do their job. Probably not reusers, once they make the initial Web query and then need deeper access. Probably not all reviewers, some of whom will want detailed information from within Smalltalk.

So, Web authoring provides increased access to end-user documenters and the merely curious, at the expense of the two usage roles most involved with development—creators and their peers!

Finally, our fourth principle of hyperliterate programming states:

The documentation for a thing must be measurable, quantitatively and especially qualitatively.

Anyone who can type "du -s" in the root directory of a UNIX Web tree will get a gross measure of the documentation, but what will that tell us? Web authoring is inappropriate for hyperliterate programming because it is file-based—you will need an entire new suite of tools to do quantitative and qualitative analysis. If you've got a perl guru in your group, that might work, but why not leverage the Smalltalk talent you've been carefully growing? It's unlikely that files of HTML code will ever be measured as part of a repository-centric metrics program.

THE WEB IS AN EXPORT TARGET

Just because we don't believe "Web authoring" is appropriate for hyperliterate Smalltalk development doesn't mean we think the Web useless. "Web authoring" means

Jan Steinman and Barbara Yates are co-founders of Bytesmiths, a technical services company that has been helping companies adopt Smalltalk since 1987. Between them, they have more than 22 years of Smalltalk experience. They can be reached at Barbara@Bytesmiths.com or Jan@Bytesmiths.com, or via http://www.bytesmiths.com.

the creation of Web documents by humans, which is great if you want to create a "cool site" or impress others with your HTML prowess, but it is at odds with the principles of hyperliterate programming, which require keeping documentation as close to the code as possible, in terms of granularity of concept, as well as physical location.

The knee-jerk problems happen when some vice-president says "We gotta get on the Web!" and the ripple effect causes otherwise bright people to do stupid things, such as dictate that all Smalltalk documentation will be in HTML.

Fortunately, HTML is easy to generate from your Smalltalk-resident documentation. If you followed the implementation sketch for Smalltalk hypertext we presented in September 1995, you already have a good start.

That implementation used a special emphasis for VisualWorks *Text* that allowed it to treat Smalltalk expressions specially. This was but a small step from "real" hypertext. We have since extended so it has a notion of both an *anchor*, or visible text with human-meaningful presentation, and an *action*, which is a block of Smalltalk source code. To distinguish this, we call it a "clickAction."

VisualWorks has been criticized for being divorced from platform capabilities, but its *Text* class provides an abstraction for styled text that is much more powerful than thoughtlessly abdicating all presentation to platform widgets. *Text* has an efficient tagged-character facility that allows you to associate arbitrary objects with runs of characters. Three common tag types are:

- A *simple* emphasis *Symbol*, such as #bold or #italic,
- a *compound* emphasis *Array* of other emphases, such as #(#bold #italic),
- a *parameterized* emphasis *Association* between a *Symbol* and an arbitrary object.

This last capability is used for things like colored text and different fonts or font sizes.

We defined a new parameterized emphasis for *Text* that contains the *Symbol* #clickAction associated with Smalltalk source code for a block. We made changes to *ParagraphEditor* so that double-clicking one of these "clickActions" causes the block to be evaluated—instant hypertext! To enable this, you need a "global method" that can discriminate clickActions:

```
Object
isClickAction
    "When used as an emphasis in a Text, do I function as
     a hyper link? Hardly!!"

    ^false

SequenceableCollection
isClickAction
    "When used as an emphasis in a Text, do I function as
     a hyper link? I do if any of my contents does."

    ^self
        detect: [:object | object isClickAction]
        transform: [:ignored | true]
        ifNone: [false]

CharacterArray
isClickAction
    "When used as an emphasis in a Text, do I function as
     a hyper link? Strings and Symbols are never
     considered active emphases. This override keeps the
     superclass method from examining each of my
     Characters to see if they are hyper links."

    ^false

Association
isClickAction
    "When used as an emphasis in a Text, do I function as
     a hyper link? I do if my key is #clickAction, in which
     case my value better be block-like, but I don't check
     that here."

    ^#clickAction == key

Text
hasClickAction
    "Do I contain any hyper links?"

    ^runs values
        detect: [:emph | emph isClickAction]
        transform: [:ignored | true]
        ifNone: [false]

hasClickActionAt: characterIndex
    "Do I have a hyper link at the given
     <characterIndex>?"

    ^(self emphasisAt: characterIndex) isClickAction
```

You may notice the strange method #detect:transform:-ifNone:, which is like #detect:ifNone:, except that when the first block answers true, the value is passed through the second block. We discovered that we usually use the result of #detect:ifNone: this way, and so made it a bit easier to do:

```
Collection
detect: booleanBlock transform: transformBlock ifNone:
exceptionBlock
    "Evaluate <booleanBlock> with each of the receiver's
     elements as the argument. Pass the first element for
     which <booleanBlock> evaluates to true through
     <transformBlock> and answer the result, or answer
     the evaluation of <exceptionBlock> if no elements
     assert <booleanBlock>."

    ^transformBlock value: (self detect: booleanBlock
ifNone: [^exceptionBlock value])
```

You should combine one or more simple presentation emphases with a clickAction for it to display differently, rather than deciding that all clickActions are going to be presented a particular way. We defined a simple emphasis #link as a blue underlined style, to make it familiar to those with Web experience. We don't have room for that code today, but you'll need to make new instance creation methods for both *CharacterAttributes* and *TextAttributes*.

BUT WHERE'S THE HTML?

Once you have the foundations—a proper object model for hypertext—spitting out HTML is almost trivial. We use a distributed responsibility pattern familiar to anyone who has examined how #printString works:

```
Object
asHtml
    "Answer a representation of myself that is suitable for
     use in a Web page. Subclasses should not override this
     method; rather, they should override htmlOn:."

    | stream |
    stream := (String new: 100) writeStream.
    self htmlOn: stream.
    ^stream contents


Object
htmlOn: aStream
    "Place on <aStream> a representation of myself that is
     suitable for use in a Web page. The default
     representation for objects is a #storeString
     representation in 'code' style. Answer <aStream>."

    ^aStream nextPutAll: '<CODE>';
        store: self;
        nextPutAll: '</CODE>';
        yourself
```

At this point, different objects are free to render themselves into HTML as they see fit. Of course, the one we've been concentrating on is *Text*, and so we have the requisite big, ugly method. Much of this complication is because of the desire to preserve some of the presentation of lines that begin with tabs. Since HTML presentation is normally driven by emphasis rather than content, treating tabs as presentation required an awkward, double-pass treatment:

```
Text
htmlOn: aStream
    "Place on <aStream> a representation of my contents
     suitable for use in a Web page. Answer <aStream>.
    For each emphasis found, write beginning and ending
    HTML tags.
    For each special HTML Character, write the appropriate
    HTML character entity.
    For lines beginning with tabs, write the proper
    indented definition list."

    | turnOff |
    turnOff := (String new: 16) writeStream.
    ^self class subscriptOutOfBoundsSignal
        handle: [:ex |
            aStream nextPutAll: turnOff contents.
            ex returnWith: aStream]
        do: [ | here tabLevel prevTabLevel thisEmphasis
        endEmph char characterEntity |
            "Handle initial tabs properly."
        Tab == self first ifTrue: [^(Text with: CR), self
        htmlOn: aStream.].

        here := 1.
        prevTabLevel := tabLevel := 0.
        "Repeat the following until I have no more data."
        [here >= self size ifTrue: [^aStream].

        "For each emphasis, build up a proper tag and an
         untag."
        thisEmphasis := self emphasisAt: here.
        thisEmphasis class == Array ifFalse:
        [thisEmphasis := Array with: thisEmphasis].
        1 to: thisEmphasis size do: [:i | | emph tags |
            tags := (emph := thisEmphasis at: i)
            isClickAction
                ifFalse: [HtmlTags at: emph ifAbsent:
                [HtmlNoTag]]
                ifTrue: [emph value asHref -> '</A>'].
            aStream nextPutAll: tags key.
            turnOff nextPutAll: tags value].

        "For lines that begin with one or more tabs, build
         a proper level of indentation."
        endEmph := here + (self runLengthFor: here).
        [here < endEmph] whileTrue:
            [char := self at: here.
            char == CR ifTrue:
                [tabLevel := 0.
                [tabLevel := tabLevel + 1.
                Tab == (self at: here + tabLevel)]
                whileTrue: [].
                tabLevel := tabLevel - 1.
                tabLevel = prevTabLevel ifFalse:
                    [char := Tab. "to suppress <P>"
                    (prevTabLevel - tabLevel) abs
                    timesRepeat:
                        [aStream nextPutAll: (tabLevel >
                        prevTabLevel ifTrue: ['<DL>'] ifFalse:
                        ['</DL>'])]].
                tabLevel > 0 ifTrue: [aStream nextPutAll:
                '<DD>'].
                prevTabLevel := tabLevel].

            "For each character with a given emphasis,
             write the character or its HTML-legal
             equivalent."
            characterEntity := HtmlCharacterEntities at:
            char ifAbsent: [].
            characterEntity == nil
                ifTrue: [aStream nextPut: char]
                ifFalse: [aStream nextPutAll:
                characterEntity].
            here := here + 1].
        aStream nextPutAll: turnOff contents.
        turnOff reset.
        here := endEmph] repeat.
    aStream]
```

If you simply feed this method to Smalltalk, it will complain bitterly, because we've added new class variables to *Text*. We don't like to change the definition of classes, but luckily, you don't need to if you're only adding class variables. We have all this code in a common ENVY application called *HyperTextBytesmiths*, with a #loaded method that adds the needed class variables to *Text* on the fly, and of course, a #removing method that removes those class variables when we're done:

```
HyperTextBytesmiths
characterEntityTable
    "Answer a Dictionary that associates non-ASCII
     characters with their HTML character entities."

    ^IdentityDictionary new
        at: Character cr put: '<P>';
        at: Character tab put: '';
        at: $" put: '&quot;';
        at: $& put: '&amp;';
        at: $< put: '&lt;';
        at: $> put: '&gt;';
        at: (Character value: 160) put: ' ';
        at: (Character value: 161) put: '&iquest;';
        at: (Character value: 162) put: '&cent;';
        at: (Character value: 163) put: '&pound;';
        at: (Character value: 165) put: '&yen;';
        at: (Character value: 167) put: '&sect;';
        at: (Character value: 171) put: '&laquo;';
        at: (Character value: 176) put: '&deg;';
        at: (Character value: 177) put: '&plusmn;';
        at: (Character value: 181) put: '&micro;';
        at: (Character value: 182) put: '&para;';
        at: (Character value: 183) put: '&middot;';
        at: (Character value: 187) put: '&raquo;';
        at: (Character value: 210) put: '&reg;';
        at: (Character value: 211) put: '&copy;';
        at: (Character value: 225) put: '&AElig;';
        at: (Character value: 225) put: '&AElig;';
        at: (Character value: 233) put: '&Oslash;';
        at: (Character value: 241) put: '&aelig;';
        at: (Character value: 249) put: '&oslash;';
        at: (Character value: 251) put: '&szlig;';
        yourself

loaded
    "Add to TextConstants."

    TextConstants
        at: #HtmlNoTag put: '->';
        at: #HtmlTags put: self tagTable;
        at: #HtmlCharacterEntities put: self
characterEntityTable

removing
    "Take away what I added to TextConstants."

    TextConstants
        removeKey: #HtmlNoTag ifAbsent: [];
        removeKey: #HtmlTags ifAbsent: [];
        removeKey: #HtmlCharacterEntities ifAbsent: []

tagTable
    "Answer a Dictionary that associates a Text emphasis
     symbol with an Association of two Strings; the key is a
     tag used to turn on the emphasis, the value is used to
     turn off the emphasis."

    ^IdentityDictionary new
        at: #bold put: '<B>' -> '</B>';
        at: #underline put: '<EM>' -> '</EM>';
        at: #Heading1 put: '<H1>' -> '</H1>';
        at: #Heading2 put: '<H2>' -> '</H2>';
        at: #Heading3 put: '<H3>' -> '</H3>';
        at: #Heading4 put: '<H4>' -> '</H4>';
        at: #Heading5 put: '<H5>' -> '</H5>';
        at: #Heading6 put: '<H6>' -> '</H6>';
        at: #italic put: '<I>' -> '</I>';
        at: #strikeout put: '<S>' -> '</S>';
        yourself
```

Now we can generate HTML from any *Text*, but it is necessarily "embeddable" HTML only suitable for the "body" part of a Web page. We have numerous ways of producing a complete page, but the most useful way works from any *ParagraphEditor*, because it is the foundation text editing class in VisualWorks. (The *Stream* implementation of #htmlFor: is left as an exercise for the reader!)

```
ParagraphEditor
htmlOn: aStream
    "Place on aStream a representation of my contents
     suitable for use in a Web page."

    ^aStream
        nextPutAll: '<HTML><HEAD><TITLE>';
        print: sensor window label;
        nextPutAll: '</TITLE></HEAD><BODY>';
        htmlFor: self text;
        nextPutAll: '</BODY></HTML>'; cr;
        yourself
```

CONCLUSION

The Web can be a powerful communication tool, but like all tools, it can be misused. Just as a screw driver or an ice pick can kill a person, mandating inappropriate use of the Web can kill a project.

We've demonstrated some techniques for exporting off-line HTML from your "hot" Smalltalk documentation. Next month, we'll show you how to serve your hot documentation "on-line," so that a Web browser can view up-to-the-minute project documentation. This should be enough to silence any VP who comes storming in, shouting "What are you guys doing about the Web?"