An essay on W3C's design principles

Table of Contents

What is a good standard?

An essay on W3C's design principles

by Bert Bos
06 Mar 2003
[translations]

Introduction

Why doesn't HTML include tags for style? Why can't you put text inside SMIL? Why doesn't CSS include commands to transform a document? Why, in short, does W3C modularize its specification and why in this particular way? This essay tries to make explicit what the developers in the various W3C working groups mean when they invoke words like *efficiency, maintainability, accessibility, extensibility, learnability, simplicity, longevity,* and other long words ending in -y.

Contrary to appearances, the W3C specifications are for the most part not designed for computers, but for people. You may think that `<h1>`, `è` and `</ul>` look ugly and unintuitive, but what would you think if they were called `A378`, `30C9` and `38F0` respectively? Most of the formats are in fact compromises between human-readability and computer efficiency. Some are more readable than others, because we expect more people to read them, others can be more cryptic, because "only" programmers will look at the source.

But why do we want people to read them at all? Because all our specs are incomplete. Because people, usually other people than the original developers, have to *add* to them. The computer will not by itself invent a new technology. Somebody will look at HTML, understand it and think of a "cool" new technology to complement it. That's how the Web (and not only the Web) progresses.

For the same reason we try to keep the specifications of reasonable size. They must describe a useful chunk of technology, but not one that is too large for an individual to understand. After all, new ideas usually come from individuals.

The most commonly heard guideline on the Web is to *separate structure and style.* It is what led to the separation of HTML and CSS into two individual specifications. And a good advice it was, too. But behind this rule there are the deeper reasons of maintainability, implementability, finding the right experts, and many more, as explained below.

The Web is all about helping humans communicate, but what hopefully becomes clear from this essay is that writing specs also is a form of human communication. There is one word that summarizes nearly all the fancy keywords of this essay, and that is:

Usability
Usability
Usability
Usability

Usability
Usability!

Maintainability

Both the specifications and the files or programs made according to them have to be maintained. There is hardly any data or service that never needs updating, moving or converting. Sometimes there is a program that can make the changes, but there always comes a time when a major change is needed and a human has to be involved. At that point it is good if the thing is of manageable size and has a fairly clear structure. After all, it may have been a long time ago since human eyes last saw it, and the original author has probably long since changed to a better job. It is probably unavoidable that, for example, an XSL style sheet is harder to change than a CSS one, but even so it is good that the designers of XSL spent some time on the right choice of keywords, provided a syntax for comments and allowed the code to be indented.

Of course, there is no help for authors (or programs!) that generate unreadable code. If you know the HTML format and you have looked at the source of some pages on the Web, you have probably wondered occasionally what monster was able to mangle the HTML code behind a simple Web page or an e-mail message in such a way that it is hard to believe that HTML was ever called a "structured language." Given the opportunity, some people will make a mess of anything.

Taking CSS as an example, CSS has many features that are there for the benefit of the people who maintain style sheets: the "@import" allows splitting large style sheets in logical units, the grouping syntax for selectors and rules allows keeping things in one place that the author expects to be changed together, shorthand properties provide a convenient (and short) way to set in a single rule several properties that usually occur together.

On the other hand, CSS stops short of even more powerful features that programmers use in their programming languages: macros, variables, symbolic constants, conditionals, expressions over variables, etc. That is because these things give power-users a lot of rope, but less experienced users will unwittingly hang themselves; or, more likely, be so scared that they won't even touch CSS. It's a balance. And for CSS the balance is different than for some other things.

But in fact, even with CSS, the power-user isn't left without means completely. There is also the DOM, that provides access to CSS through each person's favorite programming language. That's modularity.

Modularity

People can only consciously work with a limited number of concepts at any time. Short-term memory, which is what you use when solving an intellectual problem, only holds six or seven items. Thus to solve a complex problem with many variables, we divide the problem into at most six chunks and build the solution with those chunks as building blocks. Each chunk is a partial problem, that has to be solved in turn, but it may be that we can delegate it to others. When the chunks are small enough to be assigned catchy names we call them "modules."

How do you know that you have divided the problem into the right chunks? Basically by trial and error, although we have some intuitions mostly based on our language skills.

When defining modules, we try to find "logical units," things that appear to belong together. It has nothing to do with logic, but with our ability to group those things under a common nomer. Because, if we can name a group with a short name, it probably means it is a known problem. Essentially we rely on metaphors, because we get the names from other problem areas. Thus we have "passwords," "style sheets," "firewalls," "protocols" and of course "Web" (some metaphors are better than others...)

Occasionally a module turns out to be a possible chunk in another problem. Thus the syntax and selector modules of CSS3 found use in the the STTS document transformation language and PNG found application as the format for cursors and bitmaps in SVG. Some document languages use a lot of HTML, but it is debatable whether to call that the usage of a large number of very small HTML modules, or simply a variant of HTML. Probably the latter, since their functionality overlaps so much with HTML that they often sin against the principle of minimum redundancy.

Minimum redundancy

This section is called "*minimum* redundancy" and not "*no* redundancy," because, as the introduction said, our technologies are used by humans, and humans can deal with redundancy, indeed have trouble when there is none. Too much redundancy is bad for the computer (or the programmer) though, because it means implementing the same thing several times. But it is good that, e.g., you can make a paragraph in HTML red in a couple of different ways. People have different styles and different mental models, and just by choosing a different style of working they can express something subtle that helps them, even though the computer cannot (yet) deal with it.

On the other hand, XML 1.0 defines a tree structure with four kinds of leaf nodes (text strings, processing instructions, empty elements and entities). Especially since they are abstract, it is not clear why there have to be four. Why not ten? or only one?

So what is too little and what is too much? That is hard to say. The overlap in functionality between different specifications should be kept small, because it can easily lead to incompatible models that programmers will have difficulty implementing, which in turn leads to bugs. Thus SVG doesn't provide a way to encode raster images, since PNG exists for that. On the other hand, within a single specification there may well be multiple ways to do the same thing, because it is easier to keep them consistent.

Take the example of coloring a paragraph red again: CSS offers different types of rules and different expressions that all mean "red." They add very little to the implementation effort but enhance the usability of CSS a lot. On the other hand, the attribute in HTML that makes a text red is officially deprecated, because it uses a different model from CSS (no cascading) and a slightly different syntax ("red" and "#FF0000" are OK, "#F00" is not).

One warning, though: accessibility often requires that the same information is provided in several ways, e.g., both as an image and as a text. To a certain extent the two formats indeed express the "same" information, but it is information at a different level, information that is only inferred from the actual data provided. We cannot express that information, so rather than an example of the Web having *two* ways of denoting the same thing, it is an example of the Web having *no* way to express something.

Accessibility

The word "accessibility" refers primarily to the degree in which something is accessible by people with disabilities, but in a wider sense it also measures resistance to external or temporary handicaps, such as noisy environments or bad lighting.

The main method to ensure accessibility is to encode data at as high a level of abstraction as possible, but it is also important to hook into already existing accessibility technologies.

The structure/style dichotomy is a clear example of the former: rather than encoding that something is red, W3C formats allow (and urge) an author to encode first of all the reason *why* it is red. The redness is added as a rule on top of that. With HTML, for example, rather than `<font color="red">`, a careful author writes `<em class="warning">` and puts the rule that warnings are displayed in red in a style sheet. This way somebody who has no means to display (or see) red at least has a chance to substitute something else that alerts him (such as a nasty sound).

With XML-based formats, at least the well-designed ones, the problem should be less than in HTML. In contrast to some versions of HTML, most XML formats do not have stylistic elements at all. In such formats you may see an element `<warning>`, but no `<font>`, so a style sheet is obligatory.

Sometimes a whole technology can be seen as an accessibility feature. SVG has other advantages over raster image formats, but a very important point is that it allows the image to be taken apart and traversed as a hierarchy of objects, rather than a collection of colored pixels. You can in effect read out an SVG graphic. And even better if the author has inserted textual descriptions of each object.

Isn't it more work to make accessible resources? Will people not be lazy and omit structure and annotation? Not always. Some people want to be good. Some are required to be good (by law, or by their employer's public image). But it is also the case that properly structured and annotated code is easier to maintain; accessibility is sometimes an investment with an initial cost, but eventual benefit. In well-designed formats the accessibility features aren't added as an afterthought, but are merged with the features for maintainability and code structuring. An SVG graphic can be analyzed and the pieces used again to create a new graphic. A PNG image, on the other hand, is pretty much the end of the line.

The `alt` and `longdesc` attributes of HTML's `img` element *were* an afterthought. As a consequence it feels like an extra effort to have to think of an `alt` (=alternative) text. And if you think of a good one, it may not be possible, because of the limitations of HTML attributes: no way to *emphasize* a word of the `alt` text, no way to make a list or a table, etc.

The `img` element was added to HTML without thinking about accessibility (or about some other things for that matter, such as fallbacks and captions) and the alternatives that were proposed, `fig` and `object`, simply came too late and never really caught on. Accessibility features have to be integrated from the start, in the specifications as well as in the tools that implement them.

See also

[ATAG1.0]
: Treviranus, Jutta; McCathieNevile, Charles; Jacobs, Ian; Richards, Jan. *Authoring tool accessibility guidelines 1.0.* 3 Feb 2000. W3C Recommendation. URL: http://www.w3.org/TR/2000/REC-ATAG10-20000203/

[UAAG1.0]
: Gunderson, Jon; Jacobs, Ian. *User agent accessibility guidelines 1.0.* 10 Mar 2000. W3C Proposed Recommendation (draft). URL: http://www.w3.org/TR/2000/PR-UAAG10-20000310/

[WCAG1.0]
: Chisholm, Wendy; Vanderheiden, Gregg; Jacobs, Ian. *Web content accessibility guidelines 1.0.* 5 May 2000. W3C Recommendation. URL: http://www.w3.org/TR/1999/WAI-WEBCONTENT-19990505/

[XMLGL]
: Dardailler, Daniel. *XML accessibility guidelines.* 17 Feb 2000. W3C Note (draft). URL: http://www.w3.org/2000/02/NOTE-xmlgl-20000217.html

Device-independency

If a specification doesn't *have* to depend on a specific (type of) device, then it probably shouldn't. Or it should be split in a dependent and an independent part. Device-independency is in many ways the same as accessibility, although for different reasons.

A specification like CSS is somewhat device-dependent: specifying a font only makes sense on a visual medium; there is no interpretation of font on a speech synthesizer. CSS in fact divides all devices into classes and allows styles for each class to be grouped. It is different for HTML. Precisely because CSS (and XSL) takes care of the device-dependent parts, HTML can be device-*in*dependent. Thanks to that, you can do a lot more with HTML than display it on a screen (see also repurposing).

It is sometimes claimed that HTML isn't device-independent, because an HTML author assumes that his reader has a convenient medium to read text, such as a book, a magazine, a laser printer or a reasonably large screen. Reading a 5-page article on the display of most mobile phones is not really pleasant. But is that HTML's fault, or is it just that certain content is more suitable for certain environments? HTML is just as capable of expressing weather reports as novels. What is blamed on HTML is probably just shortcomings in current style sheet implementations.

Not only is it desirable to be able to use a resource on different devices at the same time, but a valuable resource should also be designed to survive future changes in technology: with technology changing as fast as it does now, a document written on an up-to-date system 4 years ago may already be unreadable, because the hardware isn't made anymore, because the software maker has gone broke, etc. Not all HTML files are worth keeping for 50 years, but HTML should at least make it possible that documents written in it are still readable for the next generation(s). Gutenberg's bible is still readable after 500 years, why should your publications be worth any less? (See also longevity.)

See also

[Rothenberg95]
Rothenberg, Jeff. "Ensuring the Longevity of Digital Documents" in: *Scientific American.* Vol. 272. pp. 42-47. Jan 1995.

[TFADI96]
Preserving Digital Information. 1 May 1996. The Commission on Preservation and Access and The Research Libraries Group, Inc.. URL: http://www.rlg.org/ArchTF/tfadi.index.htm

Internationality

To be World-Wide, the Web must be usable by people who don't speak the language of its developers. In general, it makes little sense to develop a format for the exchange of information that is only usuable for one or two human languages. Thus, if a specification allows human readable text anywhere, it must allow text in any language. That text must allow all characters from the Unicode repository and it probably should allow multiple ways of encoding them, including at least UTF-8. ASCII is not forbidden, but it cannot be the only encoding.

Don't assume that text is always horizontal, left to right and that there are spaces between the words. Not to everybody does 7/1/92 mean January 7, '92, and in many languages, 3,141 is close to pi and 3.141 is a thousand times larger.

Also, when you write a specification for the W3C, remember that W3C specifications are written in English, but mostly read by people for whom English is only a second language.

Extensibility

It would be really nice if a technology was right the first time... No different versions, no differences between applications, no need to upgrade. But in practice everything needs a version 2 and often a version 3 as well. So it is best to take this fact into account when designing version 1.

Of course, you cannot know what will be in version 2, let alone a version 3, but there are nevertheless some things you can do to make future extensions easier, in particular in the form of reserved pieces of syntax behind which you can hide new features in the future. You can make a language *forward compatible* to a certain extent in this way.

One easy trick is to add a "magic number" at the start of the file, typically including the version number, so that applications written for version 1 will recognize that a file is not the right version. That helps against misinterpreted data, but the method is rather blunt: the application doesn't do anything with data that is too new.

Some types of information are amenable to *graceful degradation,* i.e., version-1 applications can be made to approximate a desired effect within their limits, while newer applications will do the full version-2 effect.

HTML, e.g., has one simple rule that works reasonably well for text documents, and that is to treat all text in unknown elements as if it were simple inline text and to ignore unknown attributes. The developers of new versions thus have two choices, depending on whether the best fallback for a new feature is to omit the text or display it inline. HTML also has a magic number at the top of the file, but the method works well enough that browsers do not have to give up if they see a document with too new a version.

CSS has a slightly more sophisticated method, because it allows authors in many cases to include explicit alternatives for older implementations. Old programs will read what they understand and skip the rest, while newer ones will read both, and if the author has done it right, the newer features will override the older ones.

An even more sophisticated method is to provide a method for the author to specify on a case by case basis whether an approximation is acceptable or not. The C language, e.g., provides a keyword "#error" that a programmer can use to signal to the compiler that a certain feature is obligatory and that without it continuing is useless. SOAP (a set of conventions for developing restricted XML-based formats, see [SOAP1.1]) has a similar feature, as does P3P [P3P1.0].

In C, the syntax looks like this:

```
#ifndef FEATURE
#error
#endif
```

In P3P, the syntax is like this:

```
<EXTENSION optional="no">
<FEATURE.../>
</EXTENSION>
```

Where *FEATURE* is some property that is not predefined by P3P 1.0.

See also

[P3P1.0]
Cranor, Lorrie; Langheinrich, Marc; Marchiori, Massimo; Presler-Marshall, Martin; Reagle, Joseph. *The Platform for Privacy Preferences 1.0 (P3P1.0) Specification.* 10 May 2000. W3C Working Draft. URL: http://www.w3.org/TR/2000/WD-P3P-20000510/

[SOAP1.1]
Box, Don, et al. *Simple Object Access Protocol (SOAP) 1.1.* 8 May 2000. W3C Submission. URL: http://www.w3.org/Submission/2000/05/

Learnability

You sometimes hear people say that syntax is not important. Not true: syntax is one of the most important things there is. (Maybe what they really mean is that *their* syntax is better than yours...) They probably refer to the fact that there are multiple ways to write down the same model. That is true, but it by no means follows that all ways are equivalent. It is a philosophical debate whether a person’s language determines his mental model of the world or whether the language is merely the surface structure of a deeper model that is essentially the same for everybody. For human languages we leave the debate to others, but for our formal, limited languages we must follow the advice from mathematics: there is nothing like a suggestive notation for understanding the model.

The standard example from mathematics is d*t*/d*x*, which looks like (and is treated by physicists as) an infinitesimal change of *t* divided by an infinitesimal change of *x.* Mathematicians can tell you exactly in what ways it behaves like a division and it what ways it doesn’t, but the fact is that for nearly all formulas that occur in physics it *does* behave like a division. And it makes life very easy for physicist that they can simplify (d*x*/d*t*)(d*t*/d*y*) to d*x*/d*y*. It is easy because it *looks* the same as simplifying (2/3)(3/7) to 2/7. The notation for differential equations hasn’t always been like that, but it is not for nothing that all other notations are forgotten.

Learnability, then, has a lot to do with readability.

See also

[Wittgenstein86]
Wittgenstein, Ludwig. *Tractatus logico-philosophicus.* Amsterdam. Dec 1986. Athenaeum-Polak & Van Gennep. In German, with Dutch text on right-hand pages, translated by W.F. Hermans

Readability

A notation can be too short. If a seldomly used feature takes only one letter ("t"), then the few times you see the letter you will probably have to look it up, if you don’t overlook the letter completely. It is better as a complete word ("translate").

A notation can also be too long. If a keyword you use all the time takes 20 letters to type ("shapeoutlinedata"), than it could probably have been abbreviated ("d"). These examples are from SVG (which gets them right), but you can find similar cases in most languages.

Unfortunately, sometimes your best guesses turn out to be wrong. The designers of XML thought it was helpful to open and close an element with the full name of the element (`<heading>...</heading>`) and for the usages they foresaw that was indeed quite reasonable: in long text with sparse mark-up the ability to see what element you close outweighs the small redundancy [XMLgoals]. They thought allowing a shorter form (such as </> or <>) would just add to the complexity of the language. But XML is now more often used for data in which the mark-up overwhelms the content in between, and line after line of opening tag almost next to the identical closing tag is hiding the essential data by their redundancy. XML isn’t broken, but in hindsight it could have been made somewhat less expensive to use.

Of course, you can always use something else than XML. And if readability of the source is important you probably should. But it comes at a cost: you will have to think about many things that come for free when you adopt XML (how to escape Unicode characters, how to ensure unambiguousness of the syntax, what delimiters to use for nested structures, how to parse white space, etc.).

One way to use both XML and a readable format is to use converters between the two. For MathML, for example, there are several tools that allow people to edit mathematics in a familiar notation. Once there are such tools, readability of the source becomes less important. Not *un*important, though: see longevity.

CSS has its own syntax because readability is very important for a language that is used by nearly as many people as HTML itself. It could have been based on SGML, and originally there were indeed proposals for SGML-based syntaxes (see [Lie99]) and there even existed a US military standard ([FOSI]), but there was never any question that CSS syntax is better. (XML didn’t exist at the time, but it wouldn’t have helped since it is more verbose than SGML and allows fewer symbols as punctuation.)

Many other languages opt for a mixture of XML and some other syntax. XML by itself only uses angular brackets as delimiters, everything else must be done with keywords. People are generally more comfortable when other symbols, such as colons (:), semicolons (;), curly braces { } and parentheses ( ) also carry meaning. SMIL, for example, expresses times and time intervals with a notation like 3:25.5 rather than, say `<time><min>3</min><sec>25.5</sec></time>`. SVG similarly uses a compact notation for paths that is much more readable than anything with XML tags could ever be. But both SMIL and SVG still use XML for the parts where verbosity is less of a problem, thus making the job of the designers of the specification easier at an acceptable cost in readability.

And there are languages for which readability is rather less important compared to, say efficiency. PNG is an example. It is a binary format.

See also

[FOSI]
: . [Where is this thing?]

[Lie99]
: Lie, Håkon Wium; Bos, Bert. *Historical style sheet proposals.* 1997-1999. W3C. URL: http://www.w3.org/Style/History/

[XMLgoals]
: Tim Bray; Jean Paoli; C. M. Sperberg-McQueen (eds). *Extensible Markup Language (XML) 1.0.* 10 Feb 1998. W3C Recommendation. Section 1.1 URL: http://www.w3.org/TR/REC-xml#sec-origin-goals

Efficiency

According to Jakob Nielsen, people are most productive if the computer’s response to their click takes less than a second. People lose their concentration if a page takes longer than ten seconds to appear.

Of course, computers get more powerful and bandwidth increases continually, at least on average. But at the same time new devices are being connected to the Web, such as mobile phones and TVs, and they are much slower. They too will improve, eventually, although they will probably always be behind desktop computers. But then there will no doubt be new appliances (and applications). Designers of Web technology will always have to consider efficiency.

Speed loss can occur at the server, if producing a response to a query is too complex; at the network, if there is too much data; and at the client, if visualizing the data is too hard.

When designing formal languages to write resources in, designers should ensure that the languages are easy to parse and reasonably compact. "Progressive rendering" (the effect that displaying a response starts before all the data has arrived) can help as well. CSS, for example, has been designed in such a way that every line of a document can be displayed as soon as it arrives (although for text inside tables that requires some extra work by the style sheet writer). Caches help as well, and thus splitting documents in cachable and uncachable parts is a good thing. E.g., HTML and XLink allow documents to be assembled from parts, some of which may be shared among several documents. Sometimes facilities for compression or switching to a binary format may even be needed.

It is in general a good idea to provide methods in a language to remove redundancy, (although they have to be weighed against the added complexity of the language). Thus HTML and SVG have a mechanism to assign a "class" to an element and define the style for all elements of that class in a single place. SVG also allows re-use of graphical objects in multiple places: drawing a 12-point star thus only requires a single point, displayed 12 times in different places. CSS likewise has several mechanisms for grouping rules.

However, intuition isn't always a good guide when estimating efficiency. Network throughput isn't linear: a message consisting of a single TCP package is much faster than one with two packets, but four packets typically take less time than two separate messages of two packets (see [Frystyk97]). Java is often perceived to be slow, but Jigsaw (an HTTP server in Java) can typically hold its own against Apache (an HTTP server in C), because much of the perceived slowness comes from starting the Java virtual machine; once that runs, Java is fast enough (see [JigPerf]). And as a W3C study showed some time ago, most gain comes from radical changes: replacing GIF with PNG and HTTP/1.0 with HTTP/1.1 certainly helps a lot, but not nearly as much as replacing an image by some text with a style sheet. With SVG, the possibilities for that will be greatly enlarged.

See also

[Frystyk97]
: Frystyk Nielsen, Henrik; Gettys Jim; Baird-Smith, Anselm; Prud'hommeaux, Eric; Lie, Håkon Wium; Lilley Chris. *Network performance effects of HTTP/1.1, CSS1, and PNG.* 24 Jun 1997. W3C Note. URL: http://www.w3.org/Protocols/HTTP/Performance/Pipeline.html

[JigPerf]
: *Jigsaw performance evaluation.* 27 May 1998. W3C. (Part of the Jigsaw documentation, updated with each version.) URL: http://www.w3.org/Jigsaw/User/Introduction/performance.html

[Khare99]
: Khare, Rohit; Jacobs, Ian. *W3C Recommendations Reduce 'World Wide Wait'.* 1999. W3C. A summary of [Frystyk97] URL: http://www.w3.org/Protocols/NL-PerfNote.html

[Nielsen97]
: Nielsen, Jakob. *The need for speed.* 1 Mar 1997. Alertbox column. URL: http://www.useit.com/alertbox/9703a.html

Binary or text format

Most W3C specifications define a formal language for describing some type of resource: HTML describes simple text files, SVG describes vector graphics, PNG describes raster images, HTTP describes the dialog between a client and a server and URLs describe the path to a certain resource. There are exceptions, such as the several WAI guidelines, that describe meta-rules about how to design programs and specifications (a bit like this essay, in fact, but more precise...). But most people working on W3C specifications have to start with the choice: do we create a binary format or a text based one?

In most cases the answer will be "text based," because text formats allow easier bootstrapping and debugging: you can create files with a text editor, so that developing an dedicated editor or converter can wait until later; you can inspect a file to see what should have happened, in case a program does not do what you expect; and last but not least: if in 50 years or so the specification has accidentally been lost or become hard to find, there will be a chance that from looking at a few files you can reverse-engineer enough to get the essential information out again. (This is sometimes, rather optimistically, called "self-descriptive." The formats would only really be self-descriptive if every file included the text of the specification...)

Text based formats often also allow for unforeseen extensions later, because they typically have quite a lot of redundancy. Many programming languages have thus acquired conventions for putting things in structured comments or have gotten new keywords in later versions. Although binary formats do typically have some built-in extension mechanism, it is much more limited (e.g., GIF and PNG extension chunks).

But maybe the choice for text-based formats is simply made because the IETF recommends them, or because somebody in management requires XML...

In fact, binary formats aren't so bad. They are often more efficient to process and transport. They are typically smaller: the whole purpose of ZIP, MPEG and JPEG, e.g., is to use as few bits as possible. They are usually faster to process, since they need simpler parsers.

Parsers for a well-designed text-based language don't have to be inefficient, although the text-to-binary conversions typically make them slower by a certain factor. More importantly, their added complexity often leads to more bugs.

And is the lack of extensibility of binary formats so bad? Maybe not. Even if a text format is theoretically extensible, it may not be so in practice. If the extension is not backwards compatible in any useful way it is better to design a completely new format. HTML has been extended through three versions: 2.0, 3.2 and 4.0, but it cannot be taken any further without serious problems for implementers. XHTML is the successor of HTML, but is a new format, not an extension.

In conclusion: most W3C formats are text based and that will probably remain so, but don't assume that binary formats are forbidden. There are costs on both sides to be considered.

Implementability

The easier a specification is to implement, the more implementations will be made and the more compatible they will be. Easy implementation also means that implementers have time to be creative and come up with novel ways to use the technology. Implementations will be cheaper; in fact they can be virtually free if they can be made by individuals in their spare time. Fixing bugs is among the most expensive operations known to developers, especially if there are people that, unwittingly, have come to rely on them. Avoiding bugs is thus of the foremost importance. Implementability can come from simplicity or from building on well-known, existing components.

Building on existing components means using technologies that programmers are familiar with, such as context-free grammars (expressed as EBNF or some variant) or XML 1.0.

It does *not* mean adding dependencies on new or ill-understood technologies. XSLT, e.g., should probably not have used XML Namespaces. It could have distinguished between commands and data in more conventional ways. XSLT is now in danger of undergoing the same evolution towards incompatible subsets as HTML, which used SGML, another technology not understood by developers.

As Jakob Nielsen says, it is unsafe to use a new specification until at least a year after it has become official ([Nielsen00], page 34).

W3C's specifications thus far aren't very complex, at least not when compared to those of some other standards organizations, but they are already getting more intricate. W3C probably has to look at the IETF again for how to keep specs simple, because already one of the most often heard complaints is that the number of interdependencies, especially among the XML family of specs, is becoming hard to deal with, especially for developers who create their first XML-based programs (and if we want the Web to develop, we'll need many first-time developers).

Indeed, XML 1.0 itself is already too complicated for some of the applications for which it was designed, as shown by the project that has been started on the Internet to define a simplified version (see [Park00]).

The existence of two implementations of a specification is no guarantee that there will be more, but it does give a hint. It is therefore a good idea that many W3C working groups now ask for two or more implementations before submitting their work as a "Proposed Recommendation."

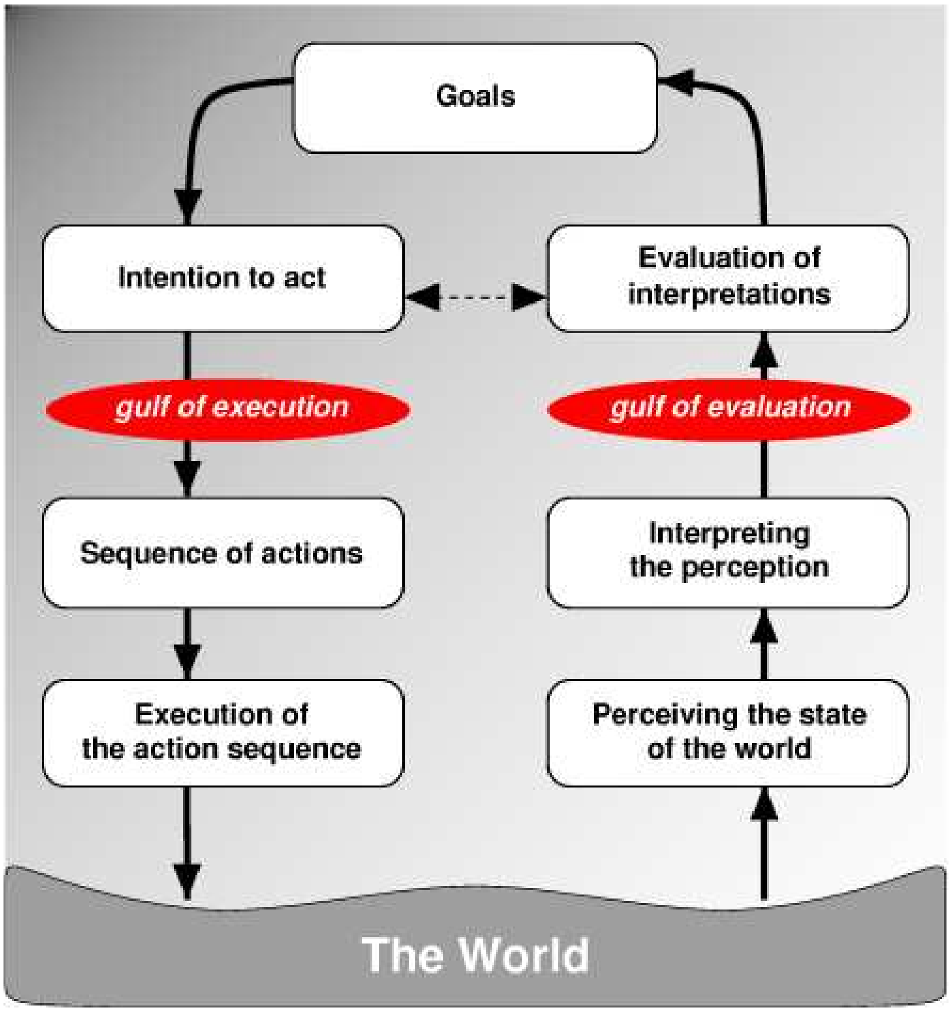
See also

[Nielsen00]
Nielsen, Jakob. *Designing Web usability.* Indianapolis, IN. Dec 1999. New Riders.

[Park00]
Park, Don (ed). *Minimal XML 1.0.* 11 Apr 2000. Draft developed on the SML-DEV [SMLDEV] mailing list. URL: http://www.docu-verse.com/smldev/minxmlspec.html

[SMLDEV]
sml-dev mailing list. 29 Nov 1999. Simple Markup Languages Developer group mailing-list. URL: http://www.egroups.com/group/sml-dev/

Simplicity



This image, which is based on one by Donald Norman, shows seven stages a person has to go through to complete an activity. If we start at the top with the person's goals, then from those goals he forms a set of intentions to change certain things in the world. He translates the intentions into a sequence of actions and executes them. He subsequently perceives the new state of the world, interprets what he sees, and compares it against what he intended to change. He may have to do another cycle if his goals were not met.

If there are tools involved, as is the case when the world is the world of the Web, then the intentions have to be translated into actions that the particular set of tools is capable of. If the way a person thinks about a problem is different from the way the tool maker did, then this translation can be difficult: think of it as trying to move from one square of a chess board to an adjacent one when all you can do is make jumps with the knight. Norman calls this step the *gulf of execution.*

The tools in our case are software and formal languages. Good programs can hide bad languages somewhat, but when the model behind the language is hard to understand or complex, such programs are hard to make. And sometimes people don't know the right tools to use.

Take an example: graphic designers want to create visuals on Web pages that consist of (in their mental model) partly overlapping images. Before there was CSS2, the only tool they had was the `<table>` element of HTML. But table cells do not overlap. However, some clever software makers were able to present an interface to them where they could put images anywhere they wanted and behind the scenes the program would cut up the images in tiny pieces and those pieces that didn’t overlap would each occupy a cell of a huge table. That worked, most of the time, but when somebody else, not possessing the author’s tool, would try to interpret what was going on, or even change the visual, he wouldn’t be able to make head or tail of it. Which brings us to the other *gulf.*

The *gulf of evaluation* on the right side of the diagram represents the difficulty a user has in mapping the visuals and messages from the system back into his own mental model. The problems here may be that the system presents incomplete or ambiguous information ("Syntax error in file") or that it employs metaphors that do not match those of the user ("green means error"). In the example above, tables were an incorrect metaphor for overlapping images.

See also

[Norman86]
: Norman, Donald A; Draper, Stephen W.. *User centered system design: new perspectives on human-computer interaction.* Hillsdale, NJ. 1986. Lawrence Erlbaum Associates.

[Norman88]
: Norman, Donald A. *The psychology of everyday things.* New York. 1988. Basic Books.

Longevity

By default, documents on the Web are written for eternity. If something is worth writing it is worth keeping. You never know when something stops being useful or who depends on it. As a consequence you should never break links. That part is well known.

But you should also write the contents in such a way that it is likely to survive changes in technology, because converting old documents may someday become too expensive. How to achieve that? Unless you can predict the future, the best rule of thumb is to stick to standards, including W3C’s specifications and guidelines.

Indeed, despite Tim’s pleas (see [TBL98]), it is unlikely that many current URLs will still work in 50 years, but the documents they referred to will still be there, and, if written in HTML, be readable as well.

That puts a responsibility on the developers of specifications. They have to always ask themselves whether the specs promote, or at least allow, Web resources to be made in such a way that they are likely to be still decipherable in 50 years time. Since developers are not much more capable of predicting the future than anybody else, they follow rules of thumb as well:

- Do not put in features that you already know will be deprecated in the next version.
- Use simple formats that can be decoded if the specification gets lost, maybe use text formats
- Be device-independent whenever possible or put device-dependent parts in separate modules.
- et cetera, see the table of contents of this essay.

And what about those broken URLs? The traditional way in which information has been kept from oblivion is by providing redundant metadata *inside* the resource: a library can catalogue a book that they found again from the information on the title page. On the Web we can actually do a little better, since we can make the metadata (partially) machine-readable. It is thus a good idea to include standard fields in each document format for information that helps identify the resource: META and LINK elements in HTML, embedded RDF in SVG, @author/@version in Java, etc.

See also

[Kahle97]
Kahle, Brewster. "Preserving the Internet" in: *Scientific American.* Mar 1997. URL: http://www.sciam.com/0397issue/0397kahle.html

[Nielsen98]
Nielsen, Jakob. *Fighting linkrot.* 14 Jun 1998. Alertbox column. URL: http://www.useit.com/alertbox/980614.html

[TBL98]
Berners-Lee, Tim. *Cool URIs don't change.* 1998. W3C. URL: http://www.w3.org/Provider/Style/URI

Backwards compatibility

There are two kinds of backwards compatibility: the obvious one of a version of a specification with previous versions of the same, and another one of new technologies with earlier ones.

Nobody forgets about the former, because there is nothing the developers of a new version know so well as the previous version they are trying to replace. Backwards compatibility is always hotly discussed.

But the latter is less obvious. It is, in a sense, the complement of extensibility and modularity. Whereas those two stress the importance of developing technology in such a way that it will work together with future new technologies, backwards compatibility stresses the importance of working well with what is already there. No new technology is designed in a void.

Not only does a new technology normally have to be compatible with earlier ones in technical matters, but also in the mental models that users have of the old technology. Introducing new paradigms always has a cost, that has to be set off against future benefits.

Take the example of CSS: people using HTML, and indeed most word-processors, are used to attaching stylistic information to concrete elements. To make style sheets acceptable at all, CSS has to allow people to continue working in the same way, while expanding their options.

As a counter-example, XSL takes a different approach to styling: people first create the style in the form of a template and then inject elements into it. It requires more motivation from users, but therefore it offers new possibilities. It aims at users that can afford to learn about recursion and loops and that need to style abstract documents that will be generated in the future (dynamic reports from a database, for example), rather than concrete articles that have already been written.

Sometimes a new technology replaces a previous one, although it is rare that it replaces the old one completely. PNG in principle is able to replace GIF completely, and XHTML can completely replace HTML. But even in such cases, some form of backward-compatibility is needed in the form of facilties for mechanical conversion of resources to the new format.

"Mechanical" means that the amount of intelligence necessary should be limited. For example, SVG partly replaces PNG, viz., in cases where PNG is used to display diagrams or graphs. Those would be better in SVG, but the backwards-compatibility of SVG with PNG is almost non-existent. Of course, in this case the benefits of SVG for images of this kind is so great that nobody complains about the lack of compatibility (but even so, SVG provides a trivial way to do the conversion: wrap the PNG in an SVG object without any interpretation).

The Web itself is designed to be backwards compatible. The URLs that give the location of resources allow for access via FTP and other protocols rather than just HTTP, and HTTP in turn allows for arbitrary types of files, not just HTML. And, as a final example, HTML allows any kind of file to be hyperlinked (via the A element) or included (by means of the IMG or OBJECT elements), not just HTML and PNG.

Interoperability

This new-fangled word is a variant of that other, rather better known new-fangled word, compatibility, and it means simply that something (a document, a program) written according to our specifications should work identically across different applications and different computers. "Identically" of course has to be qualified: you cannot display a document exactly the same on the screen of a handheld organizer as on a 19 inch color screen. But you can get the same experience and the same information from that document.

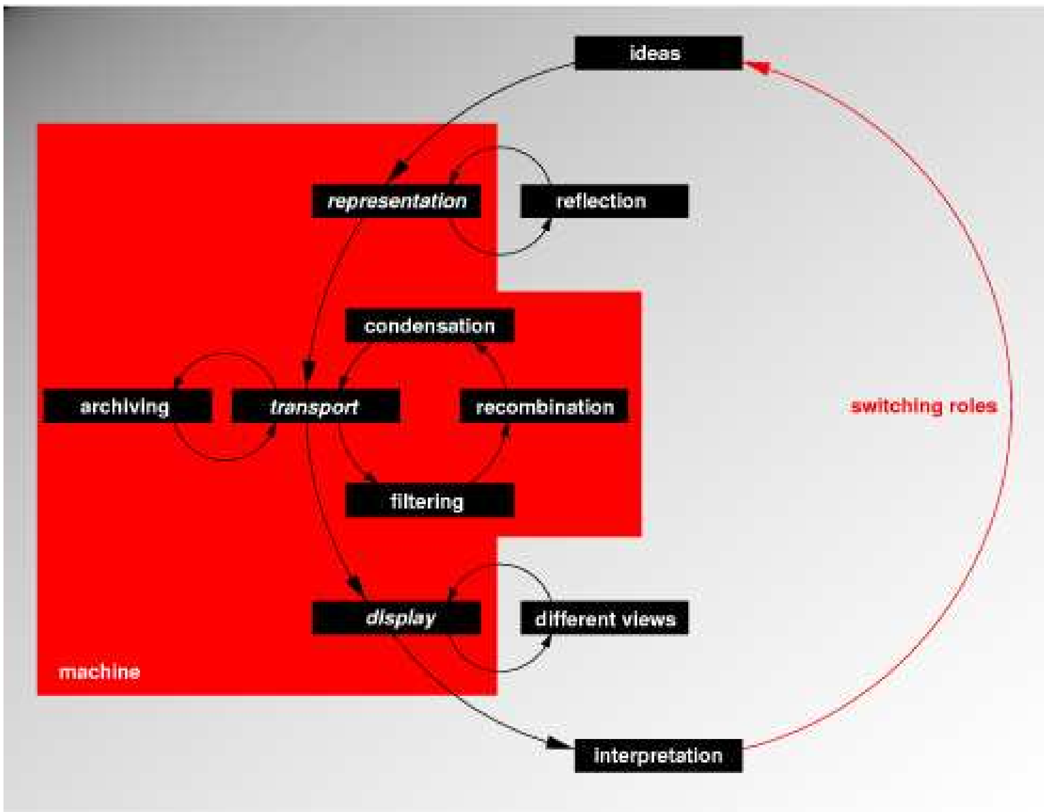
The issue is much clearer if you compare typical browsers: sure, they have different features, that's why people prefer one over the other, but you don't expect a page that looks blue in one browser to be red in another, or (what sometimes happens:) readable in one browser but not in another.

The developers of a specification, then, have to ensure that what they specify *can* be implemented by all the different implementers: it should not depend on a certain platform, it should not be open to different interpretations, and, if possible, it should be *testable.*

Often, when there are parts that can be interoperable and others that cannot (or only in a limited way), it make sense to split a specification into two separate ones, each with its own interoperability requirements.

For example, we don't expect that every program that can display static text documents (HTML) can also display dynamic presentations (SMIL) or vice versa, and thus HTML and SMIL are two independent modules.

Repurposing of content



The large circle in the diagram represents human communication with the Web as an intermediary: somebody has an *idea* (at the top); he *represents* it in a machine-readable way and enters it into the Web (the red part); the Web *transports* it and *displays* it to somebody; that somebody *interprets* what he sees (at the bottom) and may in turn become the originator of new ideas.

There are various smaller circles in the diagram, that each represent modifications of the information, hopefully enhancements, but possibly degradations. The top circle, *reflection,* is a process under the control of the author; the bottom circle, *different views,* is controlled by the reader; but the circles in between can be done by either of them, by other people, or automatically by programs such as Web spiders.

It is these smaller circles that represent the extra value that the Web can bring to human communication. But for them to work well, the original representation has to be suitable for manipulation by software.

The collective name for the manipulations done in these smaller circles is *repurposing,* i.e., the adaptation of some piece of data for a new purpose.

Timeliness

For a new technology to be successful, it has to come at the right time. Waiting too long before developing an essential module means that ad-hoc solutions will be deployed, which causes backwards-compatibility problems and high costs for the people who have to convert things to the new standards once they arive. Developing something too early may mean that it will be forgotten by the time it is needed, or worse, that it can't be used, because the intermediate steps have turned out to be different than what was expected.

But what is the right time? Hard to say. But some of the symptoms of its arrival are that people start asking for something (although it is quite an art to recognize what they are asking exactly), that there are people available with ideas, and especially that the solution appears obvious.

Use what is there

Looking back to the first 10 years of the Web, it certainly looks as if there has been a revolution. The Web now runs on HTML, HTTP and URLs, none of which existed before the '90s. But it isn't just because of the quality of these new formats and protocols that the Web took off. In fact, the original HTTP was a worse protocol than, e.g., Gopher or FTP in its capabilities, and HTML back then also wasn't quite what it is now: no embedded images, no tables, no colors...

So in the early days many people had their home pages on FTP servers rather than HTTP servers. And that fact shows nicely what made the Web possible at all: it didn't try to replace things that already worked, it only added new modules, that fit in the existing infrastructure. HTML can be served by FTP or Gopher servers; browsers can display plain text or FTP directory listings; URLs allow for all kinds of protocols, etc.

And nowadays (the year 2000), it may look like everything is XML and HTTP, but that impression is only because the "old" stuff is so well integrated that you forget about it: there is no replacement for e-mail or Usenet, for JPEG or MPEG, and many other essential parts of the Web.

Of course, technologies may get replaced by better solutions over time. GIF is slowly being replaced by PNG, SMIL replaces several other formats, ditto for SVG. The Web is in constant evolution. There is no "day 0" at which everything starts from scratch. At any time there are old and young technologies working together.

It has to be like that. Technologies improve at different speeds. Trying to release even three or four new specifications in sync is already stretching our capacities. And throwing away software that works, although imperfectly, and teaching everybody something new would be a huge waste of resources.

There is, unfortunately, a tendency in every standards organization, W3C not excluded, to replace everything that was created by others with things developed in-house. It is the not-invented-here syndrome, a feeling that things that were not developed "for the Web" are somehow inferior. And that "we" can do better than "them." But even if that is true, maybe the improvement still isn't worth spending a working group's resources on.

And especially, don't design a technology to work only with things that haven't proven themselves yet, no matter how promising they sound. The risk is too high. For example, at the moment (2000), XLink promises to become a powerful technology for adding hyperlinks to new XML formats, but CSS3, which is currently being developed and which has to style hyperlinks in XML documents cannot rely on XLink. It has to deal with non-Xlink hyperlinks as well.

See also

[Nielsen98a]
Nielsen, Jakob. *Does Internet = Web?.* 20 Sep 1998. Alertbox column. URL: http://www.useit.com/alertbox/980920.html

Design by committee

Nearly all specifications are created by a committee rather than by a single individual. The working groups of W3C typically consist of some 10 to 20 people, who work together on a new technology for a year or longer.

"Design by committee" has a bad name (specs that are a patchwork of inconsistent solutions, often redundant, and thus too big and too hard to learn), but in reality it doesn't automatically produce bad results. "Two know more than one" is another proverb, and that is exactly why working groups exist: more pairs of eyes mean more checking for errors, more creativity in finding solutions to problems, and more experience in knowing what worked or didn't work in the past.

But the problems of "design by committee" still have to be avoided. Around 15 people seems to be the limit, larger groups tend to form (informal) sub-groups and lose too much time in communicating rather than developing.

Smaller groups produce more consistent and easier to use specifications, but they may omit some things that they didn't know anybody needed. The solution seems to be to create a wider circle of interested people around them, in the form of a public mailing list.

That is how W3C develops its technologies: a working group recruited among experts, and a public mailing list for other interested people. There may be some there that have only interest in one detail or that only occasionally have time to discuss the developments. In the working group they would just have hindered the process, but on the mailing list they can give valuable contributions.

The IETF shows that it is possible to develop technologies with a single, open group of people, without distinguishing between people that have committed to a minimum amount of effort and those that haven't. But IETF groups tend to be about low-level, very technical specifications, whereas W3C specifications are (at least perceived to be) much closer to the average user and thus attract more interested people. It may also be that the environment has changed: the public on the Internet now is different from that before the Web.

Of course, the two-level system assumes a willingness of the committee to listen to the outside. It takes some time to scan the mailing list for important messages and, when needed, to answer them. But most of all it requires an openness on the part of the committee to discuss their reasons, even if they sometimes have to do with short-term company policies. And it requires a matching openness on the part of the public to accept those reasons as valid.

Expertise

If a specification is so large that there are experts on individual parts of it, but nobody wants to be called an expert on the whole, then the specification is definitely too large. Splitting it into two parts and setting up a working group for each part will almost certainly make for a better result in the end.

No amount of formal methods can substitute for an expert who is capable of holding a whole spec in his head to see the inconsistencies and redundancies.

Brevity

I assume in most of this essay that most specifications are written for programmers. That may not always be the case. The important part is to identify the **audience** and write consistently for that audience. Even if you chose the "wrong" audience, it probably still leads to more readable specs than when you try to write for "everybody."

Don’t forget that most people are short on time: reading a spec of 50 pages is just about possible, although 15 would have been better, but who is going to read 385 pages? The typical programmer will read the first 20 pages, look at a few more examples, and then start programming right away. He will fill in what he didn’t read with what he *thinks* would be the logical extension... Unfortunately, specs aren’t always logical; all too often they are the result of compromises between the committee members. Maybe that shouldn’t be the case, but that’s how it is.

Indeed, sometimes it makes sense to make a spec shorter by removing explanations, because it will mean more people will read it to the end.

In fact, removing explanations can be a good thing for another reason as well: the more explanations, the higher the chance that they contradict each other. It is better to put explanations and annotations in a separate, non-normative document, or maybe in a book.

But do not remove examples. A well-chosen example can often replace several paragraphs of explanation. And moreover they can be used as a rough test case for new implementations.

Programmers typically like to find out how things work while programming, rather than while reading English. It is a paradox: giving less explanation makes them understand better. I guess programmers like puzzles...

PS. Style sheets allow one to present alternative views of the same document, for example one with and one without the examples.

See also

[Lesch00]
Eriksson, Jan Roland; Lesch, Susan; et al.. *Friendly Specs.* 8 Apr 2000. E-mail thread on www-html [WWWHTML]. URL: http://lists.w3.org/Archives/Public/www-html/2000Apr/0062.html

[WWWHTML]
www-html mailing list. May 1994. Mailing list for discussion of HTML. URL: http://lists.w3.org/Archives/Public/www-html/

Stability

Technology changes, the future can’t be predicted, and often specification writers will have to publish something that they expect to change in the future, simply because there is an urgent need for a standard. Hopefully they will have gotten the extensibility right…

But in general, stability of a specification is a desirable characteristic. Having to re-learn how to do something is costly, creating new programs to do the same thing in a different way is costly, and converting existing documents and other resources to a different format is also costly, so changes with little or no benefit should be avoided.

The Web is a revolution. But let it not be one that eats its children. For all its shortcomings, HTML is quite a useful little format. There is absolutely no need to replace it yet.

Robustness

Networks may fail, people may make mistakes, files can get lost, software may have bugs, the new version may not be as backwards-compatible as you thought... and exactly when you desperately need certain information, you cannot get it, or it arrives damaged.

Redundancy (but not too much) can help, including the use of text files or putting a unique identification inside a document, so it can be recognized independent of its URL. And also having a printable version of a document, with enough information that the relations between documents can be followed "by hand" instead of with a computer.

Keeping things simple and not dependent on too many other technologies is a good idea. Building on top of technologies that are known to be robust helps as well.

Not just the technology should be robust, but the use of it as well. Distributing information, splitting it in smaller chunks, can help protect the different chunks against errors in some of them. On the other hand, if something is essential for the understanding of something else, the two parts should be in the same place or even in the same file, without a potentially broken network connection in between.

In very concrete terms, the above applies to the design of documents: if the document contains a picture that is cute, but otherwise not essential for understanding the text, the picture should be in a separate file, because it makes the text file shorter, less likely to contain errors, and more likely to arrive somewhere intact. But if the text contains mathematical formulas, they probably should be embedded in the document, because you cannot risk them getting lost.

The study of how to make the Web robust has only barely started (as of this writing, in 2002). But it is an important topic, because the Web is part of the infrastructure for the modern economy.