Comprehensible XML - Markup UK 2021

Erik Siegel - Xatapult - May 2021

1 Introduction

Writing software, it's all too easy to forget that there is another side to it than just "it works" or "it's fast". Most software, only throw away scripts are probably exempt, goes through a life cycle of writing, testing debugging and maintenance.

This makes it important that what you write is comprehensible, both for somebody else and yourself in a few months' time (or less). It is not just a matter of being nice: intelligible code, being able to easily grasp what is meant, reduces the chance of mistakes and bugs, and shortens development time.

Nobody is in the business of deliberately writing puzzling code. But given what we probably all encounter now and then, there is a lot of jigsaw software in the world. What is this XSLT trying to do? Why the *%$#^* is this XQuery not properly indented, so I can see what `else` belongs to what `if`? What does this variable store and what's its type? What am I supposed to pass to this parameter?

We all forget to pay attention to these kinds of things. We were hurried, hungry, caffeine-depleted, tired, or stressed. We thought this piece of software would be thrown away, but it miraculously survived for many, many years. We were in a flow and wanted to see an end-result quickly. Even although we knew that a little attention now would save many, many hours of maintenance and bug-hunting later, we did not pay enough attention.

So, what we can do about it? We have probably all seen, heard, or read something about how to write good code. We probably all to try to comply, more or less, but given what we see around us, we do not always succeed. We are *not* the computers we program; our head contains a different kind of CPU and we must deal with that.

This talk will try to provide directions, tips and tricks on how to make code more understandable. It will also provide some background on why this important and why we should try to comply, despite all the excuses we give ourselves every day. How can we do this with minimum effort. It's a mixture of things from literature and personal experience after 40 years of programming computers.

Comprehensible code is not something specifically for XML alone, so the applicability is therefore wider. Examples will be for XML programming languages.

1.1 About the author

I'm an XML specialist, doing things like consulting, designing and programming, strictly XML technology (like XSLT, XQuery, Schemas, Schematron, etc.). Before that I worked as a programmer and system architect, using languages like assembler, C, Visual Basic, PERL and several others.

Besides the technical side of things I also like communicating about it. I'm the author of two books: one about eXist-db (together with Adam Retter) and one about XProc 3.0. There are several articles I have written on https://www.xml.com.

More about me at http://www.xatapult.com.

2 Why bother?

Unless you're a super-human programmer, you've written software that contains bugs. Small and obvious ones that were found during your own test runs or more hidden ones that suddenly raised their ugly heads in production. There are entire books and conferences devoted on to how to avoid this.

Besides things in the large (design, architecture, module structure, etc.), there are also a lot of things in the small that can help to avoid or more easily detect problems. Making your code more *comprehensible* is one of them. It'll help yourself creating solid code and will be a sight for sore eyes for the maintenance people (yourself?), later on in the product's life-cycle.

You can shrug about this, but bugs can be very costly. Or worse: deadly. If you wanna have “fun”, read the Wikipedia page about bugs that matter: https://en.wikipedia.org/wiki/List_of_software_bugs. Or this one, that tries to identify the worst software bug in history: https://www.laserfiche.com/ecmblog/whats-worst-software-bug-history/. Or remember the very deadly problems with the Boeing 737 MAX, partly caused by a few lines of code.

Figure 2-1 - The spectacular explosion of the Ariane 5 flight in 1996, caused by a software bug…

Of course we don’t all write rocket, flight guidance, medical or otherwise critical stuff. What we do is usually very… non-critical, ordinary, boring, quotidian? Still, helping yourself and others to easily comprehend what a piece of software is supposed to do is not only just being nice. Some reasons for caring about this:

- Whether the software is critical or not, bugs are a nuisance. And understandable code helps detecting them. By others but also, or maybe even foremost, by yourself.
- It’s a known fact that most software spends way more time being maintained than initially written. Much of this maintenance time is spent trying to *understand* what the software is trying to do, and how. Anything we can do to help with this is pure profit.
- Software is costly, a few hours spend on comprehensibility upfront saves many, many hours later.
- Professional pride…

With my focus here on things we can do during the actual writing of the code, I’m not saying that any measures in the large don’t matter. They do, a lot. But since I spend most of my time in the (XML) software writing trenches, this is what I know most about. And since I encounter a lot of code where things are wrong (yes, also by myself), a little refresher can’t hurt.

3 What can we do?

This section contains the measures that IMHO (author’s prerogative) are most important in making code comprehensible. Most, if not all, is backed up by software engineering literature. However, what is dealt with here is subjective, both in choice of subjects, examples and proposed solutions. You might disagree or think I missed some key point. I hope nonetheless we don’t disagree about the *importance* of making code comprehensible. We probably all struggle with this and try to do the best we can. Let this all be at least be a source of inspiration and awareness.

3.1 Convention yourself

If there's one thing you can do yourself a lot of fun with, it's *conventions*. Huh, conventions? Aren’t that these long and boring lists of things you need to comply with, from variable names to comment structure? Which you never ever seem to be able to adhere to in full? Yes, that’s what I mean!

There are two main reasons why conventions are important:

- They provide more “thinking space”. What if you had to come up with a variable naming convention every time you created one? What if you had to decide how to format a function header on every occasion you started one? You don’t want that. Following a pattern, aka convention, is way easier to your working memory, already filled with the many intricate details of a program’s logic.
- It boosts comprehensibility. Being able to see what this variable name stands for, where a functions starts and ends, what `else` belongs to what `if`, helps a lot in understanding a piece of software.

I admit, conventions have a bad name. Sometimes they’re used to wrap long red tapes around programming efforts, which reduces all the fun. But applied properly they can absolutely help improving the comprehensibility of the code. How can we make them effective?

- There should’t be too many of them. It must easily fit in your head. So maybe a list of one or two pages long?
- They should’t be regarded as *absolute*. When following a convention leads to ridiculous long names/ indenting/indecipherable code/overly long functions/… (cross out what does not apply), either change the convention or decide that this is an exceptional case and break the rules.
- And yes, you have to get used to them. If the list isn’t too long and the conventions more or less make sense, they’ll soon become a habit.

With respect to comprehensibility, what are the basic things important enough to have a convention about?

- Names of variables, functions, data structures, etc.: Pick something and stick to it. I personally prefer the `lower-case-with-hyphens` convention but there’s nothing sacred about it. The `lowerMixedCase`, `UpperMixedCase`, `lower_case_with_underscores` or whatever are just as good. You can also use multiple naming styles for different things in your program, but don’make choosing the right one too complicated!

  The only thing I would definitely *not* recommend is using something without some kind of word separation. The worst example probably being `ALLCAPSWITHOUTANYSEPARATORS`: too hard to read and understand.
- Layout: Things like indentation, whitespace, empty lines, separators, comment styling, etc. Choose something and try to keep the code consistent. More about this in "The rhythm of the code" on page 3.

  You can make it easy on yourself here: find out what the pretty-print functionality of your IDE is capable of, and use *that* as (part of) your layout convention…
- Commenting: What to write comments about, how they’re formatted, etc. See "Comments? What comments?" on page 6.
- Filenames and directory structures: Again, pick some naming convention. Don’t forget to standardize the *extensions* (is an XQuery script `*.xq`, `*.xql` or `*.xquery`?). If there isn’t already one, try to invent a convenient directory structure, using standardized names, that fits the bill.

3.2 The rhythm of the code

Layout is extremely important for the comprehensibility of the code. `2+3 * 1+2` is *not* `15` (what the spacing suggests) but `7`. Or what about this coding horror:

```
<xsl:function name="local:something">
<…></xsl:function><xsl:function name="local:something-else"><…>
</xsl:function>
```

What are the things we can do to enhance comprehensibility? First a few tips about expressions and the likes:

- Use parentheses abundantly. Write `(2+3)*(1+2)` if that’s what you mean. Parentheses help to communicate the intention of your expressions.
- Choose a style about where to add spaces. Do you write: `(2+3)*(1+2)` or `( 2 + 3 ) * ( 1 + 2 )`? Format a function call like `add(1,2)` or `add(1, 2)` or `add( 1, 2 )`… Again, pick something, stick to it.
- If there’s any syntactic sugar in your language that makes expressions easier to read, by all means, use it. For instance, since version 3.0, XPath has the `=>` operator. So instead of:

  ```
  lower-case(normalize-space(translate($name, '\', '/')))
  ```

  We can now write the, IMHO much clearer:

```
$name => translate('\', '/') => normalize-space() => lower-case()
```

But what is, in my eyes, most important is the code’s “rhythm”. When you look at some script or module, can you easily see how it’s structured? Where things begin and end? What belongs to what? Here are my tips:

- Choose an indentation style and try to keep this consistent.

  For XML based languages like XSLT that's easy: indenting of XML is already (unofficially) standardized. The only thing we can bicker about is *how many* to indent and whether to use *spaces or tabs* (I personally prefer an indent of two spaces and not to use tabs).

  For text based languages like XQuery it’s a little harder. I’ve done a lot of XQuery programming lately and noticed I couldn’t come up with a consistent and simple enough indentation strategy that works in all situations. Especially for XQuery, make sure that it’s clear what code belongs to what FLWOR expression or `if/then/else` branch. Whatever you choose, it’s always making the *intention* of the code clear that matters most.

- Open some code module in your editor, scroll a bit, and look at it as from afar. Can you see how it's structured? What the building blocks are? When you’re looking for some template/function/section in the code, can you quickly find it? And in a glance see where things start and end?

  Clearly separate the main parts of your code. For instance, an XSLT program consists mainly of templates. Make it very clear where one begins and ends. Create a *sectioned layout*, like chapters/ sections in an article or book.

  I personally prefer using “comment lines” for this:

  ```
  <xsl:template …>
  …
  </xsl:template>

  <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->

  <xsl:template …>
  …
  </xsl:template>
  ```

  And use a “heavier” line in between sets of templates that belong together:

  ```
  <!-- ====================================================================== -->
  ```

  And no, it’s *not* awkward to insert (type) these kinds of lines all the time, see "Debunking some obstacles" on page 8. But you can also decide to use multiple empty lines. Or whatever works for you.

- Use empty lines and comment headers to give code inside a template/function/… a “rhythm”:

  ```
  (: Initialize: :)
  …

  (: Compute the value for … :)
  …

  (: Write it to disk: :)
  …
  ```

  Splitting code in blocks like this serves two main purposes, analogue to paragraphs in prose:

  - They force you to think more structured and do the things that belong together together.
  - Somebody which is new to the code can more easily grasp what it’s about (especially when the comment headers are helpful, see "Comments? What comments?" on page 6).

  The code blocks should’t be very long, max. 10 to 20 lines, preferably shorter. Just a single line is ok if it serves the purpose.

- Use a maximum line width to prevent your lines from overflowing/wrapping and making them hard to follow. Older books about software engineering advocate 80 characters. Given our modern big screens and the tendency not to print things, I personally prefer ~150.

  Using a line width can even flag incomprehensibility issues! When a piece of code starts regularly overflowing this, it’s usually a clear indicator the code is too complex and you’d better refactor/split it…

3.3 Names, names, names (and declarations)

Choosing the right name for something (variables, functions, templates, etc.) is extremely important. A function named `computeIt` doesn'really communicate much meaning. If we decide to call it `computeTaxForCustomer`, its intent is much clearer. Naming matters.

Here are some comprehensibility measures with regards to naming:

- Don't be shy of using long and descriptive names. We're a long way from computer languages that forbid names longer than 8 or 16 characters. Most IDEs support you by providing pop-up lists of names to choose from if you want to refer to a variable, function or template (and even if not we have copy/paste!).
- Use descriptive names. This is the absolute winner in making code self-documenting. For instance:

  ```
  <xsl:variable name="f" as="xs:double" select="($c - 32) * 0.5556"/>
  ```

  It takes a comment to tell the reader that this converts a temperature in Celsius to Fahrenheit. But if the variable names were chosen more wisely this is no longer necessary:

  ```
  <xsl:variable name="temperature-in-fahrenheit" as="xs:double"
    select="($temperature-in-celsius - 32) * 0.5556"/>
  ```

  Or even better: break out such an expression in a function:

  ```
  <xsl:function name="mod:celsius-to-fahrenheit" as="xs:double">
    <xsl:param name="temperature-in-celsius" as="xs:double"/>
    <xsl:sequence select="($temperature-in-celsius - 32) * 0.5556"/>
  </xsl:function>
  ```
- Break a long and complicated expression into smaller parts using aptly named variables. For instance:

  ```
  if (($status eq $status-success) or
    (($amount ge $amount-limit) and (not($special-account-type))))
  then …
  ```

  We can make its intent more clear if we rewrite it to:

  ```
  let $successful-attempt as xs:boolean := $status eq $status-success
  let $large-withdrawal-permitted as xs:boolean :=
    ($amount ge $amount-limit) and (not($special-account-type))
  return
    if ($successful-attempt or $large-withdrawal-permitted)
    then …
  ```

  Don't worry: only in very rare circumstances you have to be concerned about the performance impact creating a bunch of additional variables. And probably, the compiler will optimize them away for you.
- If something is in a certain unit (meters, inches, kilograms, dollars, zorkian foepies), stick the unit to the name. For an example see the Celsius to Fahrenheit conversion above.

  I specifically mention this because of a bug classic: a software mismatch between the metric system and the English measurement system caused the NASA Mars Climate Orbiter to crash in 1998, at the cost of $125 million (https://www.simscale.com/blog/2017/12/nasa-mars-climate-orbiter-metric/).
- Magic values, numeric and string constants with special meanings, should be given a name. Declaring such magic values centralizes their definition, which makes them easier to lookup and/or change and prevents bugs caused by mistyping a value. A good name clearly communicates the intent of the value.

  Here's a classic example:

  ```
  <xsl:for-each select="1 to 12">
  ```

  Why 12? Ah, this is better:

  ```
  <xsl:for-each select="1 to $months-per-year">
  ```

  It's not that I expect that the number of months in a year is going to change anytime soon. This is about communicating the intent: we're iterating over months here…

  Sometimes there is grumbling (me included) about having to create long and boring lists of magic name declarations, like this:

  ```
  declare variable $mod:status-error as xs:string := 'error';
  declare variable $mod:status-warning as xs:string := 'warning';
  …
  ```

  That looks superfluous, until you make a hard to spot "typo" bug:

  ```
  if ($status eq 'errror') then …
  ```

- A convention for building the names is also good. Things like:
  - Which abbreviations can be used (`max`, `min`, `ptr`, etc.)
  - Use something like object-action (`file-read`, `status-get`, etc.) or action-object (`read-file`, `get-status`, etc.)
  - Special prefixes for booleans (`is-`…, `do-`…).
- If you declare something, make that declaration *as complete as you possibly can*. For a variable or parameter, *always* specify it's type. For a function, always specify the return value's type. Even XSLT named templates can declare a return type (in an `as` attribute); specifying this is unusual but definitely not wrong!

  Always specifying datatypes has double benefits. It tells you more about the declaration, increasing its comprehensibility. It will also cause a whole bunch of errors to surface sooner if you make mistakes.

3.4 Comments? What comments?

Probably everyone knows the famous saying "better no comment than a wrong one", or one of its variations. And it's true: it's extremely confusing reading what a piece of software is supposed do and find out it does something different… Of course, it happens to all of us. You wrote something once, including informative comments. Then you refactor it, then refactor some more, and in having to deal with all the complexities of getting it to work, forget to update the comments.

Here are some best practices for commenting:

- Make a clear distinction between *black-box* and *white-box* comments.
  - Black-box comments are things that should be understandable without having to consult the code. For instance module, template and function descriptions. These comments must still make sense when they're sperated from the code, as for instance happens with XQuery xqDoc (http://xqdoc.org) comments (between `(:~ … :)`) in some environments.
  - White-box comments are *about* the code. These comments should help the reader to grasp what's going on.
- Do not comment the obvious, comment the *intent*. This, for instance, is completely superfluous:

```
<!-- Store the number of <thing> elements in $things-count: -->
<xsl:variable name="things-count" as="xs:integer" select="count(//things)"/>
```

  Both the (well-chosen) variable name as the expression already tell you this. This however tells the reader what the intent of the code is:

```
<!-- Add a prompt attribute to every thing for easier reporting later: -->
<xsl:variable name="things-count" as="xs:integer" select="count(//things)"/>
<xsl:for-each select="//things">
  <xsl:copy>
    <xsl:attribute name="prompt" select="position() || '/'|| $things-count"/>
    <xsl:apply-templates select="@* | node()"/>
  </xsl:copy>
</xsl:for-each>
```

- Use comments (and empty lines) for creating "rhythm". See "The rhythm of the code" on page 3.
- Use comments as the equivalent of *section titles*, even if what the comment says is obvious when you *know* the code. Remember, the reader is't always that knowledgeable.

```
<!-- Initialize: -->
…

<!-- Compute the values for … -->
…

<!-- Done, wrap up: -->
…
```

  This creates an additional, smaller, rhythm inside the bigger rhythm of templates and functions.

- Some movies on DVD (when we used to have DVDs, old-man's reminiscences) had the option to turn "director's comments" on. You then heard the movie director in a voice-over about certain scenes: how they were taken, why certain choices were made, etc. One of the best pieces of advise about commenting I ever had was that comments should be exactly like that: like a director telling you about the code.
  - Why is this code different from the rest? (bug workaround…)
  - Why this completely incomprehensible expression? (performance…)
  - What you did to make this working? (prevent problems already solved…)
  - What trap not to fall into when maintaining this code? (point out obvious mistakes…)

  Fantasizing about being a famous movie director and providing comments about your brilliant artistic choices is a constructive mindset for creating high quality comments.
- Be very careful with blocks of commented-out code. Don't make the reader guess about why you left it in. Forgotten to delete? Laziness? Carelessness? Is this still important? Can I delete it now or was there a good reason to keep it? At least provide some comment about the why if you really think it still serves a purpose.

3.5 It's the process…

In a presentation for MarkupUK 2019 called *Documenting XML structures*, I introduced the term "knowledge bubble" (https://markupuk.org/2019/webhelp/index.html#ar12.html). What I meant is that when you're busy doing something complicated, like programming, it's very hard to imagine what people on the outside don't know or will understand. You're in the middle of it now and everything is therefore understood and crystal clear.

You have to acknowledge to yourself that you're *in a knowledge bubble* during development. And being in a knowledge bubble means it's impossible to write good documentation because you don't really realize what the reader doesn't know. This is not only true for documenting things (in separate documentation), but also for writing good code comments *and* judging code comprehensibility.

So how can we overcome this obstacle? Here's how I try do this:

- I write my software with in the back of my mind that I will come back to it later. That makes it OK (and not laziness) to flag missing comments and sometimes missing non-essential pieces of code with a *TBD* marker (or whatever you want to use meaning *To Be Done*).

  Adding these TBD markers is important so you won't forget or overlook anything later. Writing a function but not feeling like adding the descriptive header comment? Add a comment with TBD. Having to write some error handling code that is not essential during initial development? Add a comment saying TBD. I think you get the picture.
- Find some time, later that week, maybe after a weekend, to review your code and fill in the TBDs. Try to do it when the knowledge bubble has deflated to a reasonable size (so not *immediately* after finishing coding).

  Even if the code is flawless (which never happens), you'll be astonished what you find with regard to comprehensibility issues.

Key is that after you *think* you're done programming, take a pause/break/weekend/vacation/cup of tea and *always* revise/review what you've written. And also: important commenting is best left to review time.

4 Debunking some obstacles

When I talk with people about this, I hear two counterarguments over and over. Let me try to debunk them, once and for all.

- "This costs too much performance…" Nonsense:
  - On our modern hyper-fast machines you won't notice those few milliseconds for creating an extra variable or performing that additional function call. If it takes time anyway, because…
  - Compilers/interpreters are smarter than you imagine and optimize a lot of your troubles away.
  - There will always be exceptions, usually loops that iterate so often you have to squeeze out the very last millisecond. If that's the case, of course, just do it. But please comment any weird, unusual or indecipherable construction lavishly, so it becomes comprehensible again.
- "XML comments, lines and stuff like that are too many keystrokes, it slows me down…". Nonsense:

  In most IDEs you can put stuff like starting a comment, inserting a line or whatever under some ctrl/alt/shift/apple combination. For instance, on my system inserting a comment is just a single keystroke away and leaves the cursor in between the `<!--  -->` markers, ready to type!

  It's a matter of finding out how this works in your environment and spend an hour or so customizing things. Well worth the effort.

5 Further reading?

There are numerous books about software engineering in general but only a few about coding itself. The ones I know of:

- Code Complete; Steve McConnell; Microsoft Press; 2004.
- Clean Code; Robert Martin; Addison-Wesley; 2009

Both still available and warmly recommended.