Item 29 Always Use a Parser

XML documents are just too rich in syntax sugar to be processed by anything short of a full-blown XML parser. I've seen many hackish systems held together by string and bailing wire based on regular expressions, grep, sed, raw stream processing, and other tools. These are extremely brittle and rarely able to handle the full panoply of documents they encounter. Problems include:

- Detecting the encoding, including handling multibyte character sets
- Comments that contain tags

- Processing instructions that contain tags
- CDATA sections
- Unexpected placement of spaces and line breaks within tags
- Default attribute values applied from the internal DTD subset
- Character references like `&#xA0;` and `&#160;`
- Predefined entity references such as `&amp;` and `&gt;`
- Malformedness errors
- Empty-element tags
- Internal DTD subsets that define default attribute values

These all have little to nothing to do with the semantic content or structure of a document. They have a great deal to do with syntax. A parser knows how to resolve all of these into the actual intended content. Very few other processes do. In fact, if you were to write your own program that handled all of this correctly, you'd be very close to inventing your own XML parser. The fact is, nothing short of a real XML parser can truly handle XML. Any program you write to process XML documents needs to sit on top of a real XML parser.

There are two main reasons developers invent their own systems based on regular expressions or other tools instead of using an XML parser.

1. They're simply not familiar with parsers and their APIs.
2. They find parsing to be too slow.

If it's simply a question of developer familiarity, the solution is simple. Learn to use SAX, DOM, JDOM, or some other API that sits on top of a parser. Numerous books can help you, including my own *Processing XML with Java* (Boston, MA: Addison-Wesley, 2002).[1]

The question of performance is more fundamental. However, fortunately it's often a canard. Before resorting to brittle non-XML tools for processing data, measure the real speed of the parser-based equivalent. Often parsing is not the bottleneck. Even if it is, the parser-based program may still be fast enough for your needs. If it isn't, you can often improve performance by moving to a different parser. For instance, Piccolo is often noticeably faster than Xerces, though it's not quite as feature rich. The slowdown may be the parser's fault but not the API's. A different parser with the same API may well do better. If it is the API's fault, you may be

1. See http://www.cafeconleche.org/books/xmljava/ for more information.

able to switch to a different API that performs better on your class of documents. (Items 32 and 33 discuss which APIs are appropriate for which tasks.) Finally, you may be able to live without some optional features like external entity resolution and validation that increase the cost of parsing.

However, let's assume that it is indeed the parser's fault. You're using the fastest API and parser available, and you still can't get the performance you want. Is it then acceptable to write a quick and dirty program that saves time by skipping a lot of mandated well-formedness checks and not processing all the syntax sugar? Is it acceptable to write your own mini-parser that properly handles only a subset of XML? I think the answer is no, it is not acceptable. I tend to side with Bertrand Meyer here. Although not specifically addressing XML, his more general point is correct:

> Necessary as tradeoffs between quality factors may be, one factor stands out from the rest: correctness. There is never any justification for compromising correctness for the sake of other concerns such as efficiency. If the software does not perform its function, the rest is useless.[2]

Developers think they can get away with compromising correctness because they assume they know the input format. They know the documents will always be well formed. They know all the element names in advance. They know the documents don't use CDATA sections, document type declarations, or processing instructions. Sometimes, as in SOAP, this is even required by the specification.

Nonetheless, relying on such assumptions is dangerous. In a heterogeneous, distributed, network environment, it's insane. Sooner or later (and more likely sooner) these assumptions will be violated. SOAP messages are sent with processing instructions, the specification not withstanding. Authors do use character and entity references even when they're told not to. Programmers put in document type declarations for testing and then forget to take them out in production. An upgraded library may begin inserting character and entity references whereas before it used literal characters. Any syntax that can be used will be used, and programs need to be ready for this.

Often developers object that they're only using the XML documents internally, on their intranets. These are never passed through the firewall.

2. Meyer, Bertrand. *Object-Oriented Software Construction,* 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1997, p. 15.

Thus they have absolute confidence that the documents will always adhere to the constraints their homegrown systems require. These developers have been fortunate enough never to work with Wally or have a pointy-haired boss, but sooner or later we all have to deal with Wally. Assume nothing! Verify everything, even if you're only on an intranet. Sooner or later somebody or something is going to violate your assumptions.

At the absolute extreme, documents are passed between two well-tested and debugged computer processes on the same computer that never talk to anybody else. The output of one process is tied very closely to the input of the other. No human ever intervenes and the code is never changed; or if it is changed, it's only changed in sync with the other system. In this case, it seems perfectly reasonable to make additional assumptions about the format of the data being read. For instance, if you know the sending process never generates comments, you don't need to write the code to handle them. Indeed, if there were such processes in the real world, this might be true. However, in practice nothing is ever so clean. It may not happen today, it may not happen next week, it may not happen before you jump ship to a company with fewer pointy-haired bosses, but sooner or later the sending process is going to change the documents it sends. Perhaps this will happen because the new programmer who took your place is modifying the system but managed to misplace all the detailed documentation you left behind. (And if you aren't the sort of programmer who leaves behind documentation, they have an even bigger problem.) It may happen after a library is upgraded, and the new version uses entity references instead of character references or just puts in a comment identifying itself as the generator of the XML document. It may even happen because some programmer is using telnet to manually insert documents into the system to figure out what it does. Do you want to tell your CIO that because your program didn't use an XML parser, it missed a well-formedness error in the input data and consequently the database running all the stores in the tri-state area was corrupted and crashed at 1:22 P.M. on Christmas Eve?

Hopefully by now you're convinced that you just can't do better than a real XML parser. But what should you do if your systems are still too slow? I suppose you could always throw hardware and memory at the problem. Sometimes that's enough. However, you may reach a point where you have to admit that XML is not the right approach for your system. If you really do have an unfixable performance problem, you might need to consider using a simpler format that requires less work from the parser, such as tab-delimited text. This loses many of the well-known benefits of XML, but if you're considering throwing away XML syntax and well-formedness rules to gain speed, you've lost those already. What you're processing may look like XML, but it isn't, not really. However, this doesn't happen often. Most systems can optimize the XML parsing to the point where it is no longer a crippling deficiency.